Frederic T. Chong
Christoforos Kozyrakis
Mark Oskin (Eds.)

LNCS 2107

# Intelligent Memory Systems

Second International Workshop, IMS 2000
Cambridge, MA, USA, November 2000
Revised Papers

Springer

Lecture Notes in Computer Science 2107
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
*Berlin*
*Heidelberg*
*New York*
*Barcelona*
*Hong Kong*
*London*
*Milan*
*Paris*
*Singapore*
*Tokyo*

Frederic T. Chong   Christoforos Kozyrakis
Mark Oskin (Eds.)

# Intelligent Memory Systems

Second International Workshop, IMS 2000
Cambridge, MA, USA, November 12, 2000
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Frederic T. Chong
Mark Oskin
University of California
Dept. of Computer Science
Davis, 95616 CA, USA
E-mail:{chong,mhoskin}@cs.ucdavis.edu

Christoforos Kozyrakis
University of California
EECS Computer Science Division
415 Soda Hall 1776
Berkeley, 94720-1776 CA, USA
E-mail:kozyraki@cs.berkeley.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Cognitive technology: instruments of mind : 4th international conference ;proceedings / CT 2001, Warwick, UK, August 6 - 9, 2001. Meurig Beynon ...
(ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ;
Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2117 : Lecture notes in
artificial intelligence)
ISBN 3-540-42406-7

CR Subject Classification (1998): B.3, B, C, D.4, F.3

ISSN 0302-9743
ISBN 3-540-42328-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg
Printed on acid-free paper SPIN 10839540 06/3142 5 4 3 2 1 0

# Preface

We are pleased to present this collection of papers from the Second Workshop on Intelligent Memory Systems.

Increasing die densities and inter-chip communication costs continue to fuel interest in intelligent memory systems. Since the First Workshop on Mixing Logic and DRAM in 1997, technologies and systems for computation in memory have developed quickly. The focus of this workshop was to bring together researchers from academia and industry to discuss recent progress and future goals.

The program committee selected 8 papers and 6 poster session abstracts from 29 submissions for inclusion in the workshop. Four to five members of the program committee reviewed each submission and their reviews were used to numerically rank them and guide the selection process. We believe that the resulting program is of the highest quality and interest possible. The selected papers cover a wide range of research topics such as circuit technology, processor and memory system architecture, compilers, operating systems, and applications. They also present a mix of mature projects, work in progress, and new research ideas.

The workshop also included two invited talks. Dr. Subramanian Iyer (IBM Microelectronics) provided an overview of embedded memory technology and its potential. Dr. Mark Snir (IBM Research) presented the Blue Gene, an aggressive supercomputer system based on intelligent memory technology.

Several people contributed to making this workshop happen. We would like to thank the members of the program committee for the considerable time they spent during the review and selection process. David Patterson (UC Berkeley) and Mark Horowitz (Stanford), the steering committee members, provided valuable advice on the scope and the organization of the workshop. We would also like to thank Larry Rudolph (MIT), James Hoe (CMU), and the rest of the ASPLOS-IX organizing committee for their help with local arrangements, registration, financing, and the proceedings. Finally, we would like to thank all the authors that submitted their papers to this workshop.

May 2001 Fred Chong, Christoforos Kozyrakis, and Mark Oskin

# Workshop Committee

## Co-chairs

| | |
|---|---|
| Frederic Chong | (University of California at Davis) |
| Christoforos Kozyrakis | (University of California at Berkeley) |

## Steering Committee

| | |
|---|---|
| David Patterson | (University of California at Berkeley) |
| Mark Horowitz | (Stanford University) |

## Publicity and Publications

| | |
|---|---|
| Mark Oskin | (University of California at Davis) |

## Program Committee

| | |
|---|---|
| Krste Asanovic | (Massachusetts Institute of Technology) |
| John Carter | (University of Utah) |
| Frederic Chong | (University of California at Davis) |
| Nikil Dutt | (University of California at Irvine) |
| Jose Fortes | (Purdue University) |
| John Granacki | (USC Information Sciences Institute) |
| Patrick Hanrahan | (Stanford University) |
| Peter Kogge | (Notre Dame University) |
| Christoforos Kozyrakis | (University of California at Berkeley) |
| Konrad Lai | (Intel) |
| Kazuaki Murakami | (Kyushu University) |
| Josep Torrellas | (University of Illinois at Urbana-Champaign) |
| Woodward Yang | (Harvard University) |

# Table of Contents

# A 64Mbit Mesochronous Hybrid Wave Pipelined Multibank DRAM Macro

Junji Ogawa[1] and Mark Horowitz[2]

[1] Fujitsu Laboratories of America
jogawa@fla.fujitsu.com
[2] Computer Systems Laboratory, Stanford University

**Abstract.** This paper describes a high bandwidth and low latency hybrid wave-pipelined data bus scheme for multi-bank DRAM macros on single chip multiprocessors. Long data bus lines inserted with multiple wave-pipelined stages at each bank input/output are further divided by periodically inserted synchronizing registers to overcome cycle time degradations due to skew and jitter effects in the wave-pipe. Each memory macro controller controls the access sequence not only to avoid internal bank access conflicts, but also to communicate with the other controllers through the hybrid bus. A SPICE simulation result is shown assuming for a 64Mbit macro comparing four 128bit wide data bus schemes. The hybrid scheme can realize over 1GHz on-die data bus for multi-bank DRAM.

## 1. Introduction

Designers have long known that the growing gap between DRAM performance and processor performance would limit system performance. With the continued scaling of process technologies, the large available silicon die area allows for the integration of DRAM and logic onto a single die. This increases the DRAM bandwidth and decreases the latency [2][3]. But with the possibility of putting multiple processors on a single die, the pressure on the DRAM bandwidth further increases [1][9][10].

Multi-banking the DRAM [14] can help ease this pressure by allowing multiple non-conflicting requests to access the DRAM concurrently [5][6]. However, even with multi-banked embedded DRAM, the connection between the processing elements and DRAM macros can still be a bottleneck. This paper proposes a repeated, wave-pipelined data bus [7][8] for multi-banked DRAM macros. The proposed data bus scheme breaks the long bus lines into multiple repeated stages with a synchronizer in the middle, thus making the bus delay linear with length. We can have multiple requests or replies in flight on the line at the same time by wave-pipelining the bus. Using wave pipelining we avoid the additional delay from synchronizing register elements [4]. Jitter on the clocking line and process variation ultimately limits the length of the bus, but inserting a synchronizer in the middle alleviates this problem at the cost of some additional delay. The periodic insertion of registers represents a hybrid between a fully synchronous bus and a fully wave-pipelined bus. The proposed hybrid wave pipeline method using multiple banks can meet the high bandwidth requirement in future SOCs and multiprocessors.

## 2. Concept of the Hybrid Wave Pipeline

Figure 1 shows the concept of the DRAM data path hybrid pipeline scheme. Two macros are shown in figure 1, and each macro is composed of a memory controller (MC) and a DRAM macro, which consists of *k* independent banks, each with *m*-bit wide data input/output ports. Figure 1 shows how multiple DRAM macros can be shared between different processors or memories. Two multi-bank DRAMs are connected together using their backside ports. Each macro has two dedicated clock wave pipeline data paths inside, and pipe expansion is enabled through the backside connection using synchronizers to share the macros between different requestors such as a CPU or other IP macro. A system clock needs to be provided only into each memory controller and into the mesochronous synchronizers. Each of the two requestors can access all banks up to the *addressable bank boundary*. The addressable bank boundary is set up statically before memory access operations begin. One of the two requestors can access memory banks beyond the physical boundary between macros through the synchronizer.

The purpose of hybrid wave pipelining is to balance high bandwidth and low latency cost-efficiently on the data path between the memory controller and memory banks. For future multiprocessor systems having multiple access requestors to a memory macro, the wide I/O port giving high bandwidth without loss of latency is a desirable feature of the on-die macro.

Assuming multi-banking is implemented, there are many ways to achieve a high bandwidth. For example, an easy way is simply to give a wide port to each bank and making the whole memory macro I/O wider. However, getting higher bandwidth in this manner is costly in terms of the physical wiring area. Moreover, the bandwidth is available only to CPUs and IP macros designed to handle it, for example, a special

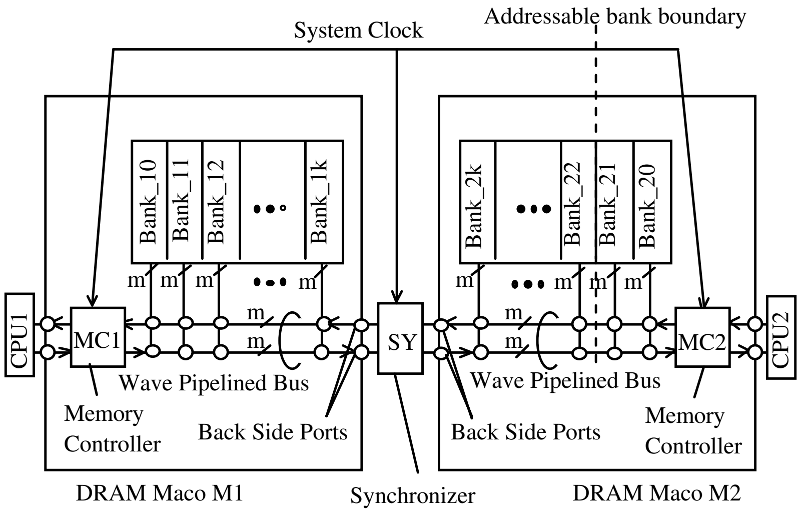


**Fig. 1.** Concept of Hybrid Pipeline Scheme; Each of two DRAM macros, M1 or M2, has *k* banks and a memory controller (MC). Outgoing and incoming data between MC and banks travel on the wave-pipelined bus. The opposite end of the wave-pipelined bus connects to a synchronizing register clocked by a system clock.

purpose streaming-data machine [1][9][12][18]. Another way to achieve high bandwidth is to use a mainstream high-speed I/O technique in the standard DRAM such as RAMBUS [3][13][16], DDR or SLDRAM [17], and for the embedded macro data paths as well. However, these techniques are optimized for board-level data buses where wiring is extremely tight, and moreover, they would require an expensive PLL in each RAM bank if they were implemented on-chip.

It is easier to increase wires and banks on-chip than on a PC board, but there are still trade-offs required to harmonize latency and bandwidth, while recognizing that wires are not free [13]. The proposed hybrid wave pipeline scheme is a candidate solution that maximizes the bandwidth per wire while maintaining short latency and reducing the cost of communication paths between different access requestors in future large VLSI chips.

Before describing the hybrid wave pipeline scheme in detail, in the next section we analyze the latency and area trade-offs for different bank sizes and IO widths.

## 3. Analysis of Latency and Cost of Multi-banked DRAM

One of the crucial parameters in designing multi-banked DRAM macros is the bank size. While using smaller banks to get high bandwidth decreases the bank access time and usually the macro access time, the area overhead increases. Assuming layout using a 0.18um embedded DRAM cell technology and 256Mbit total on die memory, Figures 2 and 3 show estimated area and latency versus bank granularity. The base design is for a macro composed of 8 x 32Mbit banks using a 16bit port. Figure 2 shows the normalized area of a 256Mbit DRAM macro for a variety of bank sizes using 16bit, 128bit or 256bit access port each. The figure shows that the area increases sharply as the bank granularity is decreased to get high bandwidth, especially for the wide I/O banks.

Figure 3 plots the bank and macro access time tRAC against the bank size. For the bank sizes shown, both the bank and macro access times improve with decreasing bank size mainly due to shorter word and/or bit lines. A smaller bank granularity and a wider bus achieve higher bandwidth, and one can also reduce the random access latency, tRAC, by 20% compared to the base design by tripling the macro size with 128 or more I/O ports. However, from the cost-performance point of view, 1Mbit or 2Mbit bank size with 128-bit or 256-bit in a 0.18um cell technology are good alternatives, balancing the bandwidth and latency while paying a much lower (~1.5X) area penalty.

Using at 0.18um DRAM cell technology, approximately 256Mbit of DRAM can be implemented on a single die. Based on the above estimation, it is a good choice to divide the memory on die into 128 x 2Mbit banks with a 128-bit width. There's still flexibility in selecting the number of macros to divide the 256Mbit memory into, corresponding to the number of access requestors on die, each of which needs an independent controller.

## 4. Data Bus Design Options

### 4-1. Fully Synchronous and Asynchronous Bus

Figure 4(a) shows a conventional data bus (CBUS) connecting all of the banks using dedicated read and write buses. This method has been widely used for a long time in actual standard DRAM designs due to the area savings by sharing wires physically for the bus line with all of the banks. As technology scales, sharing the bus wires with banks in a

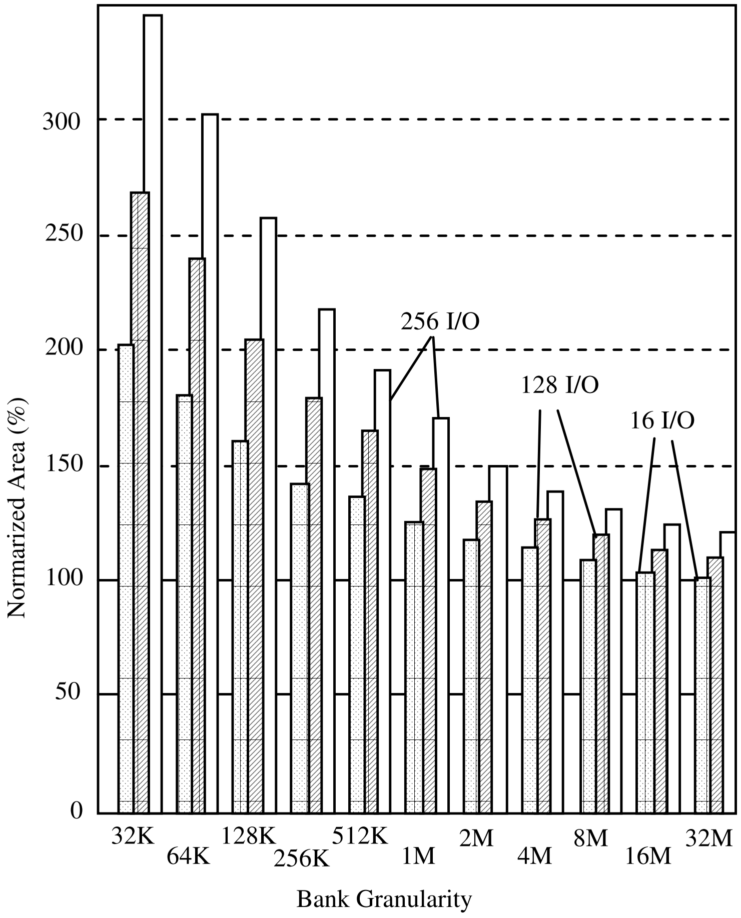


**Fig. 2.** Bank Granularity vs. Estimated Macro Area (normalized.) The base design is a 256 Mbit DRAM divided into 8 banks with a 16-bit wide ports. The three kinds of bars indicate 16-bit wide, 128-bit wide and 256-bit wide port macro respectively.

large part of the chip makes it difficult to keep up with high data rate required from DRAM. This scheme needs thicker upper layer metal wires for the bus to reduce resistance, and also spacing between bus wires must be relaxed to reduce capacitance and crosstalk noise. The end result is serious area cost and performance degradation for future large chips.

Figure 4(b) shows a fully synchronous pipeline scheme (FSP) [4], also using dedicated read and write buses. In this scheme, both buses are pipelined using registers at each bank I/O. The FSP scheme has the advantage of higher bandwidth than the conventional scheme, however the latency of the FSP bus is larger and increases with pipe depth, since each data token is advanced only one segment per clock cycle. Another advantage of FSP is the ease of increasing its maximum frequency as long as the system clock is distributed with small skew. Finally, the FSP scheme can use finer metal pitch (lower layer metals) than the conventional bus scheme because the paths between registers, corresponding to the pipe stages, are smaller.

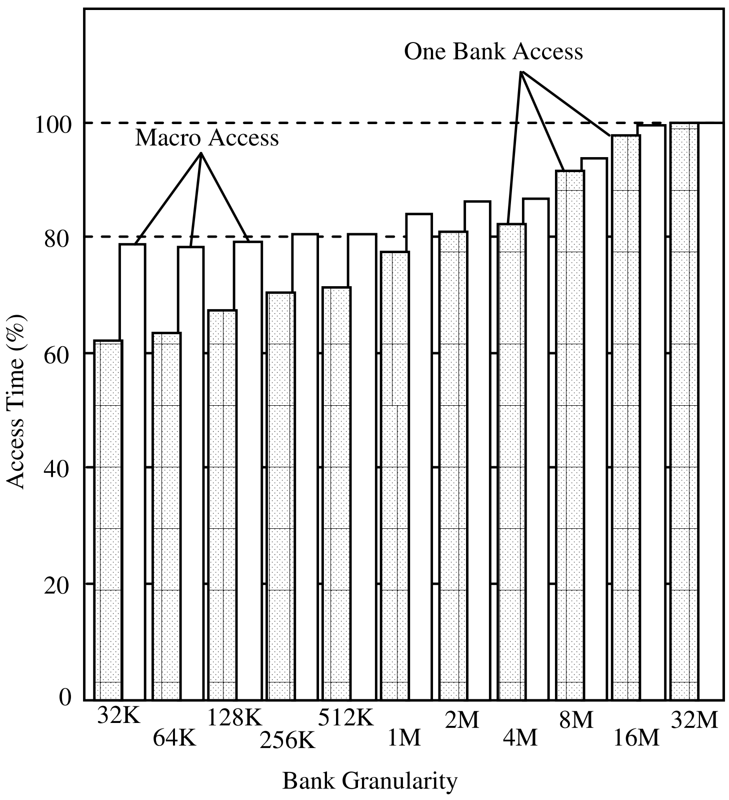


**Fig. 3.** Bank Granularity vs. Normalized Access Time (tRAC)

Another choice is a fully asynchronous bus. In general, a fully asynchronous bus can't achieve higher bandwidth nor can it achieve lower latency in real designs due to the time overhead for the inevitable handshaking between stages. On the other hand, a wave-pipeline method with mesochronous synchronization has been proposed as a possible solution to achieve bandwidth close to the fully synchronous scheme and latency close to the conventional scheme [7][8]. However, wave-pipelining design is difficult due to the critical skew and jitter control problem, and the difficulty increases as the design scales up. A longer depth wave-pipeline increases skew due to dynamic voltage and temperature variation and on-chip process variation.

## 4-2. Hybrid Scheme Based on Dedicated Clock Wave-Pipeline

To overcome the problems discussed in the previous section, we propose a hybrid pipeline scheme (HBP) based on a dedicated clock wave-pipelined bus (CWP), in which wave-pipeline segments are connected together periodically with synchronizers.

Figure 5(a) shows the concept of the CWP scheme. Each bus is divided into multiple segments, each of which corresponds to one bank I/O. Instead of using registers in every stage as in the FSP scheme, we wave-pipeline the bus using combinational logic blocks at each bank to buffer the data. Tx and Rx denote clocking signals. Tx is a dedicated clock used for the write bus, and Rx is a dedicated clock for the read bus. Rx and Tx can be generated independently, as long as they have the same frequency. As seen in figure 5(a),

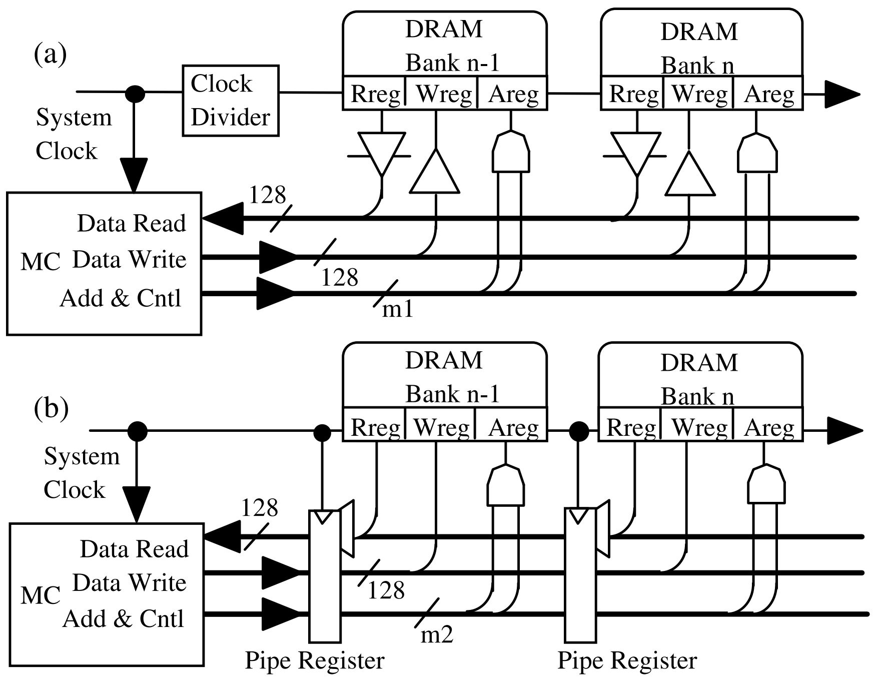


**Fig. 4.** (a) Conventional Bus (CBUS), (b) Fully Synchronous Pipeline Bus (FSP)

if the macro is not connected to another macro through a synchronizer, it is possible for the Rx signal to be generated by the return path of the Tx signal at the end of the wave pipe opposite the memory controller.

Both clocking signals propagate along replica delay lines, which match the delay of the data lines. The memory controller includes a mesochronous synchronizer to receive read data associated with Rx and then synchronize the data to the system clock domain. The clocking signals Tx and Rx facilitate proper latch timing at the inputs and outputs of the DRAM banks. The mismatch between Tx and Rx at any memory bank doesn't cause a problem provided the memory controller is programmed to insert appropriate null data cycles when sending read/write requests so that the actual write and read latch timings at the bank I/O are spaced by greater than the minimum bank cycle time. In a given cycle, only the timing relationship between Tx and write data or between Rx and read data needs to be controlled.

Figure 5(b) shows the hybrid pipeline scheme, HBP. In the HBP scheme, a set of synchronizers and registers are inserted between two buses, each of which has multiple wave-pipeline stages. Synchronization adds at most one additional system clock cycle to the data between wave-pipeline stages. In the HBP scheme, the load of the system clock can be greatly reduced compared to the FSP scheme. The maximum numbers of wave-pipe stages is determined roughly by limiting the maximum skew generated in the pipeline to less than ½ of the clock period.

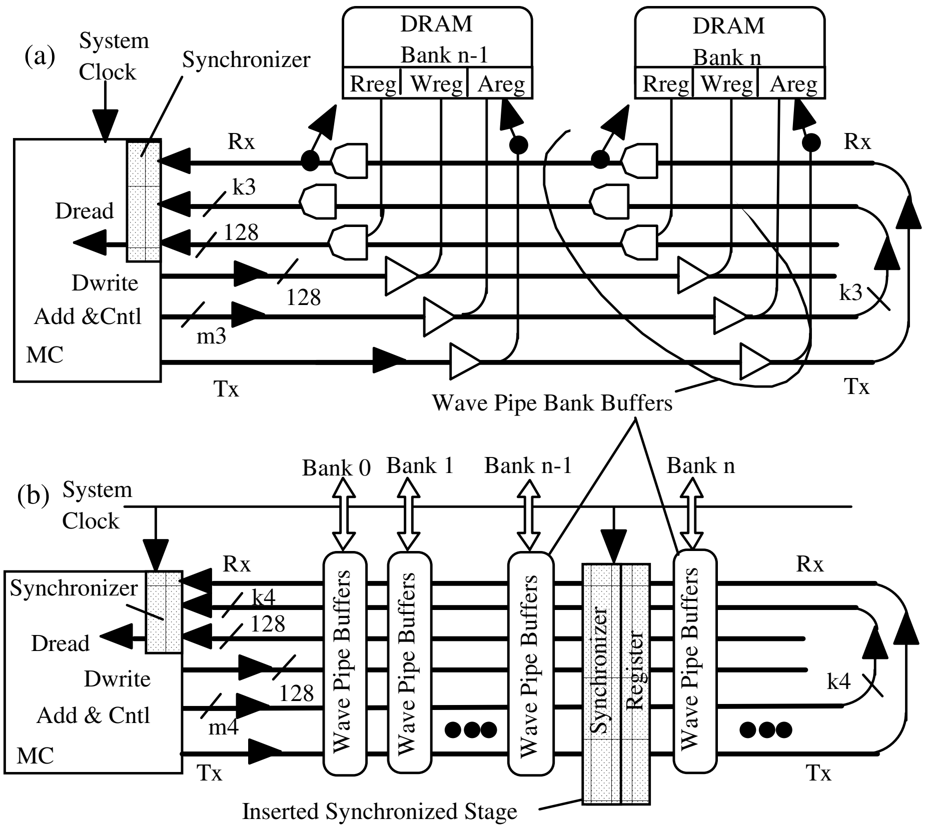


**Fig. 5.** (a) Dedicated Clock Wave Pipeline Bus, (b) Hybrid Scheme

## 4-3. Hybrid Circuit and Stage Operation

Figure 6(a) shows the unit circuit of the wave-pipeline. An or-and (OA) gate receives both traveling data from the previous neighbor stage, *PathIn,* and outgoing data from the bank, *Rdout*, and generates the next stage in the wave-pipeline, *PathOut.* The write input of the bank, *Wdin*, is buffered from *PathIn*. The static signal *Boundary_signal* is used to shut off the pipeline at an addressable memory boundary. The wave-pipeline stage has negligible area penalty compared to a CBUS, due to its use of the simple OA repeater component. Assuming the same wire pitch used for the bus line, the area penalty of the CWP bus compared to the CBUS is less than 10%. The total macro area penalty is less than 0.6%, assuming the bus occupies 6% of the memory macro. Usually, a CBUS wire is wider than a CWP bus wire, so the area penalty will be reduced in an actual design.

To latch write-in data and addresses correctly into a RAM bank, each uses its local Tx clock, which can also be used to activate the bank. On the other hand, the MC needs a mesochronous controller to receive its 128-bit data correctly. The internal MC synchronizer needs an initializing operation to learn the pipe delay characteristics. Individual banks don't need synchronizers, even while sending read-out data after bank activations triggered by Tx, because the MC knows when the write or read enable signal should be sent, and how many null data cycle should be inserted to skip a cycle.

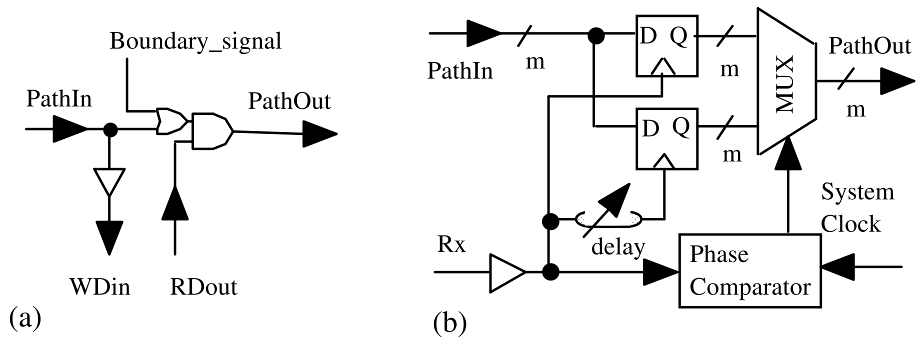


**Fig. 6.** Hybrid scheme Unit Circuit: (a) CWP bank write data-in buffer and OA, (b) Synchronizer

Figure 6(b) shows a simple embodiment of a mesochronous synchronizer, which has 2 flip-flops for 2-way interleaving and a multiplexor or 2x2 switch for every data line, and one shared phase comparator and adjustable delay element. If the frequency is high, more interleaving flip-flops are needed with finer gradations of clock phase. Dynamic phase adjustment and an accurate phase-interpolator should be provided in the synchronizer for fine-tuning. Although synchronizers both in the MC and in the middle of the wave-pipe cost additional layout area, the penalty is less than 1% of the macro area due to the relatively small number needed in the HBP scheme.

Figure 7 shows the CWP read path circuit and the bank read data registers in detail. A timing diagram of read operation in Figure 7 is shown in Figure 8. Before starting the operation, all lines are pre-charged to “1” and transition only when sending a “0”. The dedicated reference clock, Rx, always travels on the replica wires in the same direction as outgoing data at the same time, as shown in the figure, so that data and clock (Rx) propagation delay between Pathin (N04)-N06-N14 and RxIn (N00)-N01-N10 are matched to minimize skew.

R0d, R1d and R2d denote 128-bit wide read data registers in three banks 0, 1, and 2, respectively. The registers are clocked by their local Rx signals (N02, N12 and N22). Outputs data N03 from R0d goes to N05, which is one of two inputs of the wave-pipe OA gate (the OR gate for the Boundary-signal is not shown in the figure for simplicity). Clock in to data out delay between RxIn (N00)-N02-N03-N05-N06 and N01-N10-N12-N13-N15-N16 should be matched as well. In this way, the maximum frequency of the read operation is limited mainly by the sum of flip-flop delay (Tsetup + Tck-q) plus data and clock skew, as long as the inserted synchronizer can follow the frequency.

Since high bandwidth requires a large number of wires in the wave pipeline, reducing the area of the bus is an important issue in an actual design. We can wire the bus using the intermediate metal layers for data lines, which typically have twice the resistance of the top metal layers. Since over half of the delay is contributed by skew and flip-flop related timing such as clock to Q delay at maximum bandwidth, the delay contributed by wires is not dominant in the hybrid scheme. Therefore, it is possible to reduce the area penalty of the bus by using the intermediate metal layers without significantly increasing latency. The resultant macro overhead of this HBP scheme is estimated to be around 2%.

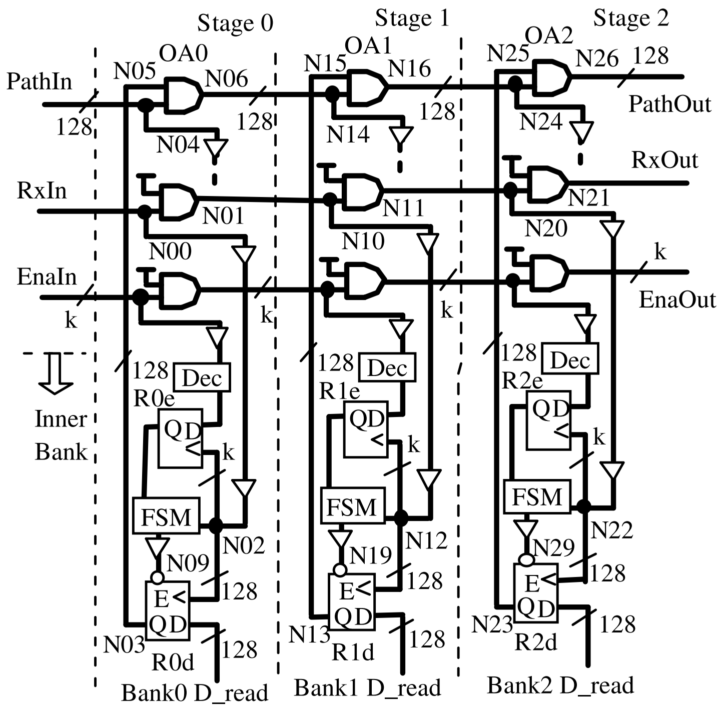


**Fig. 7.** Wave Pipeline Circuit for Data Read in HBP and CWP

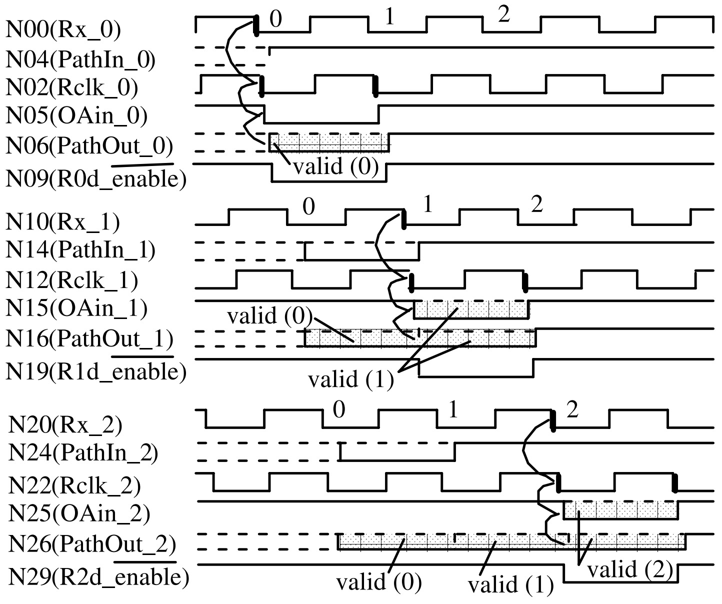


**Fig. 8.** Timing Diagram of the Wave Pipeline Circuit (see. Figure 7)

### 4-4. Comparison of Bandwidth and Latency

Figure 9 shows bandwidth and latency comparisons for the four bus schemes, CBUS, FSP, CWP and HBP, each with 128-bit input/output and 2Mbit banks. For the simulations, we used a 0.18um CMOS logic circuit technology with 0.18um DRAM cell technology. (We expect future processes aimed at SOC will combine high-density DRAM cells with high-speed CMOS transistors for logic.)

Figure 9(a) and 9(b) results were obtained using SPICE simulation. In our SPICE simulations, we modeled the power supply distribution using an RC mesh, and forced a dynamic voltage variation by attaching piece-wise linear noise current sources at each bank representing the bank transient current. The noise is initially adjusted to cause a 5% p-p jitter on the path in each 3-sigma worst-case process corner in order to quantify skew and performance degradation. The system clock was assumed relatively clean with only a fixed estimated skew of 120psec.

Assuming use of intermediate metal layers for the bus lines and 2Mbit bank size, the fully synchronous bus, FSP, has four times higher bandwidth than that of the CBUS when the pipeline depth is over eight, as shown in Figure 9(a). However, under the same conditions, the FSP has at least 30% higher latency than the conventional scheme, as shown in Figure 9(b). Wave pipelining in CWP can achieve both high bandwidth and low latency so that the hybrid scheme, HBP, can realize bandwidth close to FSP and latency close to CWP even with the longer length bus, as long as synchronizers are inserted periodically.

The HBP advantage will increase depending on the bus length and the degree to which future wire technologies can scale RC characteristics. In fact, the wave pipeline itself has a length limit depending on the process variation and jitter within each pipe region for a given clock frequency. This limit can be reduced by: 1) using simple OA-buffers in replicated layout for both data and dedicated clock lines as explained above, and 2) inserting a synchronizer in the middle of the path.

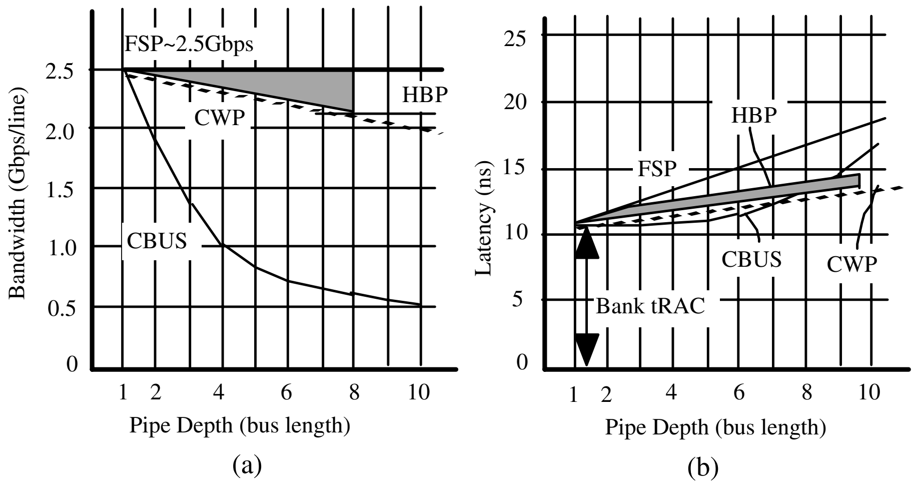


**Fig. 9.** (a) Bandwidth vs. Pipe Depth, (b) Latency vs. Pipe Depth

### 4-5. Taking Advantage of the Latency Difference between Banks

The MC controls access sequences [5][10][11] when conflicting bank addresses arrive. The MC design could be enhanced to manage the latency difference of each bank and pass the data to the requester when it is ready. However, to take advantage of the lower latency to nearby banks, a handshake is required between the access requestor and the memory macro to know when the response from the memory returns. Advanced multiprocessors, including stream specific machines, generally use deep-pipe processing, and it seems to be rather difficult to take advantage of these kinds of address dependent latency differences without stalling their pipelines frequently, especially when the difference in total latency from different banks is small. In future large chips, if the area ratio between the processing units and the DRAM becomes smaller and the ratio □tRAC/tRAC becomes larger, it will become more desirable to take advantage of the latency difference.

## 5. Embedded Macro Architecture

### 5-1. 64Mbit Macro Architecture

Figure 10(a) shows the block diagram of a 64Mbit macro; four 64Mbit macros are on a die. This 64Mbit macro has 32 x 2Mbit banks and two streams of wave pipe paths. Figure 10(b) shows the 2Mbit bank in detail, which is divided into 8 blocks having 1K sense-amplifier, 256 word lines, 128-bit read and write data buffers, address registers, and control register and logic.

In Figure 10(a), two streams of wave pipeline path, p00 to p07 and p10 to p17, are shown. Each pij denotes one pipe stage data path circuit block shared between two banks. For example, the stage p00 is used for both bank #00 and bank #08. An internal memory controller manages access sequence control not only to arbitrate access requests from different source [5], but also to manage the wave pipelining to prevent bank-to-bank access conflicts. The macro can have two ports, a main port on the front side near the MC, and a sub-port on the backside at the far end of the wave pipeline. In the HBP scheme, the backside port connects to the synchronizer (not shown in Figure 10.)
In terms of the simultaneous access to one macro from multiple requestors, the number of accessible ports is equal to the number of wave-pipeline streams designed into the macro.

### 5-2. Four-Tile Configuration on a Single Chip

Figure 11 shows a configuration example of four 64Mbit macros, M1-M4, using a hybrid pipeline with synchronizers connecting M1 to M2 and M3 to M4. If each memory controller has additional logic, M1 can connect to M3, and M2 can connect to M4 vertically on the left and right sides in this figure.

By connecting two banks at their backsides as shown in Figure 1 and 11, and inserting the synchronizer, the wave pipeline becomes expandable. As the path length increases, which means a deep pipe, this hybrid scheme gains an advantage compared to both FSP and CWP. The wave pipeline itself has a length limit depending on the process variation and jitter within each pipe region, but inserting a synchronizer reduces this limit.

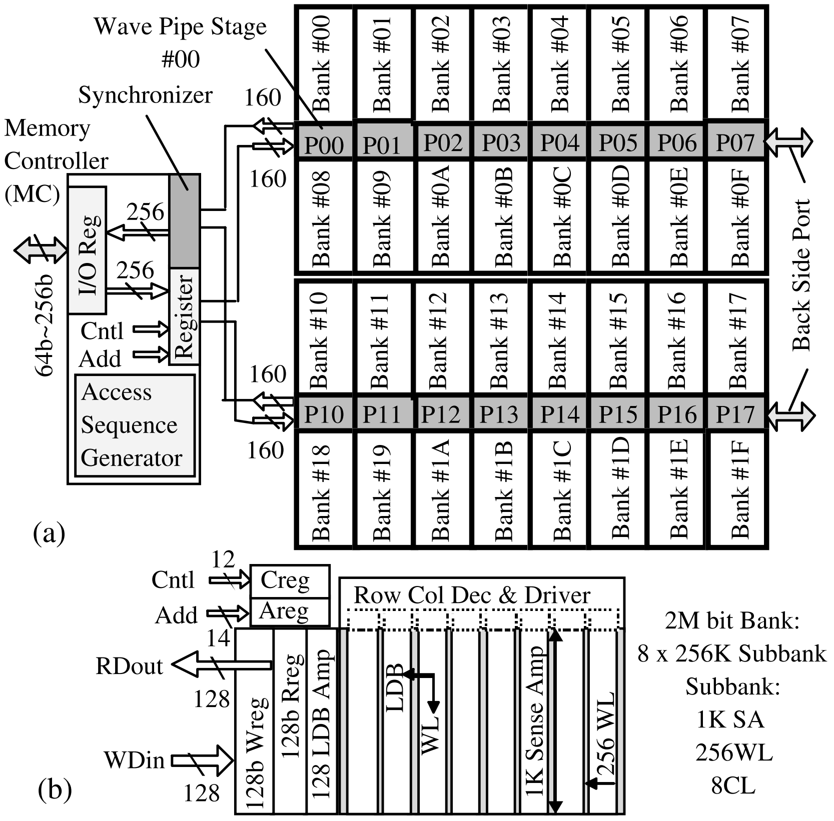


**Fig. 10.** Block Diagram of a 64Mbit DRAM Macro. (a) 64Mbit Macro, (b) 2Mbit Bank

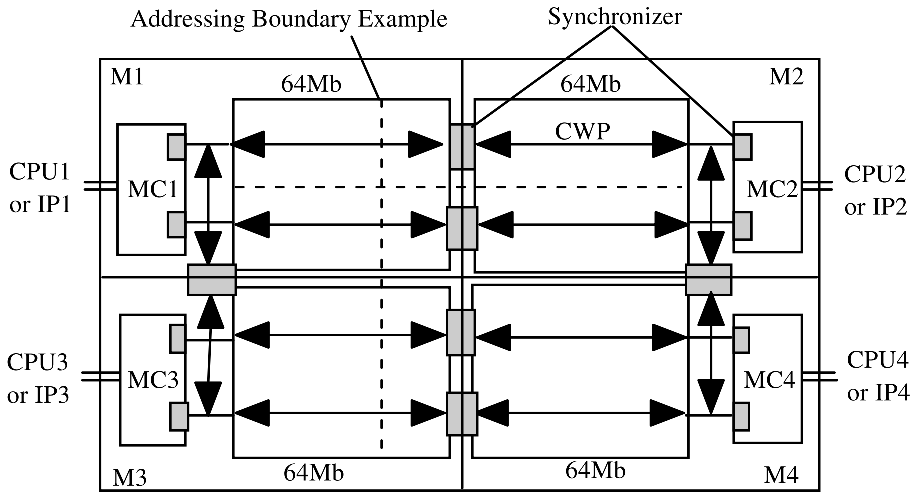


**Fig. 11.** Example Block Diagram of Four-Tile Configuration

## 6. Conclusion

Future systems will demand higher bandwidth and lower latency from DRAM. Moving to embedded DRAM improves both of these metrics, but the on-chip bus to the DRAM can still be a bottleneck. We have proposed a hybrid bus scheme employing wave-pipelining and periodic synchronization to achieve a bandwidth near that of a fully synchronous bus and latency near that of a conventional bus.

## 7. Acknowledgements

The author thanks Dr. Ken Mai and Dr. Ron Ho of Stanford Computer Systems Laboratory, Dr. Yasuo Hidaka of HAL Computer Systems Inc., and Mr. William Walker of Fujitsu Laboratories of America for useful discussions, comments and encouragement.

## References

[1] [MaiISCA'00] Ken Mai, et al., "Smart Memories: A Modular Re-configurable Architecture," *In proceedings of the 7th Annual International Symposium on Computer Architecture*, pages 161-171, June 2000.
[2] [PattersonMicro'97] David Patterson, et al., "A Case for Intelligent RAM," *IEEE Micro*, pages 34-44, Mar./Apr. 1997.
[3] [CrispMicro'97] Richard Crisp, "Direct RAMBUS Technology," *IEEE Micro*, pages 18-28, Nov./Dec 1997.
[4] [TakahashiISSCC'00] O. Takahashi, et al., "1GHz Fully-Pipelined 3.7ns Address Access Time 8kx1024 Embedded DRAM Macro," *In Digest of Technical Papers ISSCC 2000*, Pages 396-397, Feb. 2000.
[5] [WatanabeISSCC'99] T. Watanabe, et al., "Access Optimizer to overcome the 'Future Walls of Embedded DRAMs' in the Era of Systems on Silicon," *In Digest of Technical Papers ISSCC 1999*, Pages 370-371, Feb. 1999.
[6] [KimuraISSCC'99] T. Kimura, et al., "64Mbit 6.8ns Random Row Access DRAM Macro for ASICs," *In Digest of Technical Papers ISSCC 1999*, pages 416-417, Feb. 1999.
[7] [YoonJSSC'99] Hongil Yoon, et al., "A 2.5-V, 333-Mb/s/pin, 1Gbit, Double-Data-Rate Synchronous DRAM," *IEEE Journal of Solid-State Circuits*, Vol.34 No11, pages 1589-1597, Nov. 1999.
[8] [ManVLSI'96] Jin-Man Han, et al., "Skew Minimization Techniques for 256-Mbit Synchronous DRAM and Beyond," *1996 Symposium on VLSI Circuits Digest of Technical Papers*, pages 192-193, June 1996.
[9] [OluktonISCA'99] K. Olukton, et al., "Improving the performance of speculative Parallel Applications on the Hydra CMP," *In proceedings of the 1999 ACM International Conference on Supercomputing*, June 1999.
[10] [RixnerISCA'00] Scott Rixner, et al., "Memory Access Scheduling," *In proceeding of the 7th Annual International Symposium on Computer Architecture*, pages 128-138, June 2000.
[11] [Panda'99] Preeti R. Panda, N. D. Dutt, A. Nicolau, "Memory Issues in Embedded System-On-Chip –Optimization and Exploration-," Kluwer Academic Publishers, ISBN 0-7893-8362-1, 1999.
[12] [RixnerISM'98] Scott Rixner, et al., "A Bandwidth Efficient Architecture for Media Processing," In Proceedings of the 31st Annual International Symposium on Microarchitecture, pages 3-13, Nov.-Dec. 1998.
[13] [RAMBUS-website] http://www.rambus.com/developer/quickfind_documents.html
[14] [Mosys-data sheet] http://www.mosysinc.com/prelease3/, MC80364K64, 64Kx64 PBSRAM.

[15] [HorowitzSRC'99] Mark Horowitz, et al., "The Future of Wires," *SRC White Paper: Interconnect Technology Beyond the Roadmap*, 1999, (http://www.src.org/cgi-bin/deliver.cgi/sarawp.pdf?/areas/nis/sarawp.pdf).

[16] [MukaiISSCC'00] Hideo. Mukai, et al., "New Architecture for Cost-Efficient High-Performance Multiple-Bank," *In Digest of Technical Papers ISSCC 2000*, Pages 400-401, Feb. 2000.

[17] [ParisISSCC'99] Lluis. Paris, et al., "A 800MB/s 72Mb SLDRAM with digitally-Calibrated DLL," *In Digest of Technical Papers ISSCC 1999*, Pages 414-415, Feb. 1999.

[18] [GealowJSSC'99] Jeffrey C. Gealow, et al., "A Pixel Parallel Image Processor Using Logic Pitch-Matched to Dynamic Memory," *IEEE Journal of Solid-State Circuits*, Vol.34 No6, pages 831-839, Nov. 1999.

# Software Controlled Reconfigurable On-chip Memory for High Performance Computing

Hiroshi Nakamura[1], Masaaki Kondo[1], and Taisuke Boku[2]

[1] Research Center for Advanced Science and Technology, The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo, 153-8904, Japan
{nakamura,kondo}@hal.rcast.u-tokyo.ac.jp
[2] Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1 Tennodai, Tsukuba-shi, Ibaraki, 305-8573, Japan
taisuke@is.tsukuba.ac.jp

**Abstract.** The performance gap between processor and memory is very serious problem in high performance computing because effective performance is limited by memory ability. In order to overcome this problem, we propose a new VLSI architecture called SCIMA which integrates software controllable memory into a processor chip in addition to ordinary data cache. Most of data access is regular in high performance computing. Software controllable memory is better at making good use of the regularity than conventional cache.
This paper presents its architecture and performance evaluation. In SCIMA, the ratio of software controllable memory and cache can be dynamically changed. Due to this feature, SCIMA is upper compatible with conventional memory architecture. Performance is evaluated by using CG and FT kernels of NPB Benchmark and a real application of QCD (Quantum ChromoDynamics). The evaluation results reveal that SCIMA is superior to conventional cache-based architecture. It is also revealed that the superiority of SCIMA increases when access latency of off-chip memory increases or its relative throughput gets lower.

## 1 Introduction

Processor performance has been improved drastically by clock acceleration and ILP (instruction-level parallelism) extraction techniques. Main memory performance, however, has not been improved so much. This performance disparity called memory wall problem[1] is very serious. To solve this problem, cache memory is widely used. However, cache is not effective in large scientific/engineering applications[2] because data set is much larger than cache capacity. In high performance computing (abbreviated as HPC hereafter) area, although future advancement of semiconductor technology will certainly enlarge on-chip cache size, the whole data set can never reside in data cache because the data size itself grows in proportion to processing ability.

The ability of memory system is characterized by two factors, latency and throughput. There have been proposed a lot of latency hiding techniques, such as larger cache line, prefetching, lock-up free cache, and so on. However, all of these latency hiding techniques lead to increase of the pressure on memory bandwidth. Because the bandwidth of off-chip memory will not grow as rapid as processor performance, reducing off-chip memory traffic is essentially required.

Therefore, it is important firstly to exploit temporal locality and making good use of wide on-chip cache bandwidth. Secondly, decreasing the number of off-chip memory accesses is helpful for reducing performance degradation caused by long memory access latency.

As for the first issue of exploiting temporal locality, cache blocking [3] is a well-known and promising optimization. Unwilling line conflicts, however, still increase memory traffic and degrade performance. To solve this problem, good tile size selection algorithm [4] and padding technique [5] have been proposed so far. However, programs should be rewritten carefully depending on the detail of both the cache structure and the data array structure. Moreover, these techniques cannot completely remove line conflicts among different data arrays. For example, they cannot avoid the unfortunate but frequent situations where data with little reusability pollutes data cache and flushes out other data which will be used soon.

As for the second issue, the number of off-chip memory accesses can be decreased by making the size of each data transfer large. Adopting larger cache line could be one solution. However, unnecessary data transfer must be avoided because it wastes memory bandwidth which is the most valuable resource. Thus, for non-consecutive data access, this solution is harmful.

The essential reason for these problems is that it is by far difficult for hardware to control data location and data replacement. Because most of the data accesses in HPC (High Performance Computing) applications are regular, it is reasonable to control data location and replacement by software. Thus, we propose a new VLSI architecture named *SCIMA: Software Controlled Integrated Memory Architecture*. SCIMA integrates software-controllable addressable memory into processor chip as a part of main memory in addition to ordinary cache. Hereafter, we call that memory “On-Chip Memory” (As opposed to that, we call off-chip main memory “Off-Chip Memory”).

Since On-Chip Memory is explicitly addressed by software, only the required data is transferred into the On-Chip Memory without flushing out other required data. Unfortunate conflicts can be avoided. In this point, On-Chip Memory is better at exploitation of temporal locality than cache. Therefore, SCIMA has the potential to solve the problems of cache and achieve higher performance.

## 2 SCIMA

### 2.1 Overview

Fig. 1 shows the schematic view of the proposed architecture *SCIMA*. In SCIMA, addressable On-Chip Memory is integrated into the processor chip in addition to ordinary cache.

Location and replacement of data are controlled by software explicitly in On-Chip Memory, whereas those are controlled by hardware implicitly in cache. We

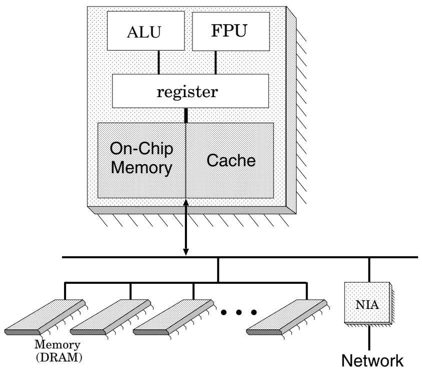


**Fig. 1.** Overview of SCIMA

employ SRAM as the On-Chip Memory. Since our target is HPC applications, the whole data cannot reside in On-Chip memory even if DRAM is used. Thus, we give higher priority to speed rather than capacity. Cache is still provided to work for irregular data accesses.

## 2.2 Address Space

On-Chip Memory occupies one consecutive part of logical address space. We assume On-Chip Memory is much larger than ordinary page. Then, frequent TLB misses degrade performance seriously if On-Chip Memory is controlled by ordinary TLB. Therefore, On-Chip Memory is treated as a large page. Two special registers are introduced to identify the On-Chip Memory area as shown in Fig. 2.

– On-Chip Address Mask Register (AMR): This mask register indicates the size of On-Chip Memory. If the least significant $m$ bits of AMR are 0, On-Chip Memory size is $2^m$byte.
– On-Chip Address Start Register (ASR): This register holds the beginning logical address of On-Chip Memory. This address must be aligned to the multiple of On-Chip Memory size.

The following equation tells whether the given address is within the On-Chip Memory area or not.

$$if\ (a\ given\ address\ \&\ AMR) == ASR\ \ then\ \ On{-}Chip\ Memory\ area \qquad (1)$$

**Inclusion Relation Between On-Chip Memory and Cache**

All the address space has a cacheable/uncacheable property. This property is managed by TLB and page-table mechanisms like the ordinary current processors. In SCIMA, the On-Chip Memory space, which is not under the control of

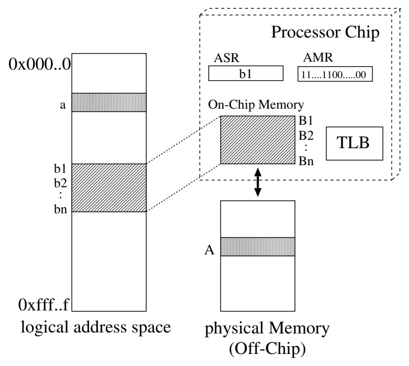


**Fig. 2.** Address Space

TLB, is always handled as uncacheable. Therefore, there is no inclusion relation between On-Chip Memory and cache.

## 2.3 Data Transfer Among Memory Hierarchy

The following two kinds of data transfers are available.

– register $\leftrightarrow$ On-Chip Memory $\leftrightarrow$ Off-Chip
– register $\leftrightarrow$ cache $\leftrightarrow$ Off-Chip Memory

Ordinary load/store instructions invoke transfers between registers and On-Chip Memory when the accessed address is within On-Chip Memory area. Otherwise, they access cache as usual and invoke line transfer when cache misses.

**page-load and page-store** Data transfers between On-Chip Memory and Off-Chip Memory are invoked explicitly by *page-load* or *page-store* instructions which are newly introduced. Notice that the term of *page* is different from ordinary page used in virtual memory. In this paper, *page* is a data unit transferred by one *page-load* or *page-store*. The size of page is assumed to be several KBytes. The source/destination addresses and the size of data transfer are identified by these instructions. These instructions can specify block-stride data transfer which can pack non-consecutive data of Off-Chip Memory and transfer into a consecutive area of On-Chip Memory. This is helpful for effective use of limited On-Chip Memory area and Off-Chip bandwidth.

## 2.4 Reconfiguration of On-Chip Memory and Cache

Total memory size which is available within a processor chip depends on semiconductor technology and the number of transistors devoted to the processor core. It is a difficult problem to decide the best ratio of cache and On-Chip Memory

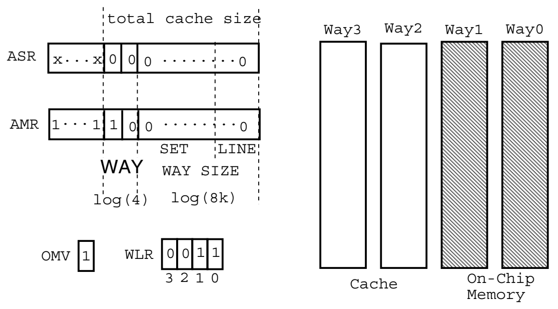


**Fig. 3.** Example of On-Chip Memory and Cache Configuration

sizes under these constraints. The answer highly depends on the characteristics of target applications. Thus, we propose a reconfiguration mechanism where On-Chip Memory and cache share the hardware memory structure and the ratio of them can be changed on the same hardware. Due to this feature, SCIMA is upper compatible with conventional memory architecture. This subsection shows the hardware mechanism.

**Hardware Mechanism**

In addition to ASR and AMR described in section 2.2, we introduce the following special registers for reconfiguration of On-Chip Memory and cache.

– Way Lock Register (WLR): The bit width of this register is equal to the degree of cache associativity (the number of ways): If the bit of the corresponding way is set to on, that way is locked as On-Chip Memory.
– On-chip Memory Valid(OMV): This register has 1 bit entry which indicates whether any way is utilized as On-Chip Memory.

Fig. 3 shows an example of configurations. This figure illustrates the case where 32KB 4way cache is divided into 16KB 2way cache and 16 KB On-Chip Memory. On-Chip Memory area is assigned by a system call. When the system call is executed, data in the corresponding ways is flushed out, the corresponding bits of the WLR and OMV are set to on, and the right values are set to AMR and ASR. The WAY bits of AMR indicate the number of ways locked as On-Chip Memory and the WAY bits of ASR indicate the way from which On-Chip Memory is allocated. In Fig. 3, the WAY bits of AMR are 10 and those of ASR are 00. This indicates that two ways are utilized as On-Chip Memory and locked beginning from way 0. Table 1 shows the possible configurations of On-Chip Memory in the example of Fig. 3[3] .

[3] Note that if OMV is 1, WLR is determined by the WAY bits of ASR and AMR. Otherwise, all the bits of WRL is 0. Therefore, exactly speaking, WRL is redundant information.

**Table 1.** Configuration of On-Chip Memory

| On-Chip Memory size | WAY bits of ASR | WAY bits of AMR | Ways utilized as On-Chip Memory | WLR | OMV |
|---|---|---|---|---|---|
| 32KB | 00 | 00 | way0,1,2,3 | 1111 | 1 |
| 16KB | 00 | 10 | way0,1 | 0011 | 1 |
| | 10 | 10 | way2,3 | 1100 | 1 |
| 8KB | 00 | 11 | way0 | 0001 | 1 |
| | 01 | 11 | way1 | 0010 | 1 |
| | 10 | 11 | way2 | 0100 | 1 |
| | 11 | 11 | way3 | 1000 | 1 |
| 0KB | — | — | N/A | 0000 | 0 |

**Actions of Memory Access**

When a memory access occurs, the accessed address is checked whether it is within the On-Chip Memory area or not by using ASR and AMR (equation 1). If the access is for On-Chip Memory, the WAY part of the accessed address bits indicates the way to be accessed. In the example of Fig. 3, if the WAY part of the accessed address is 00, Way 0 is accessed. The important point is that the sets to be accessed are determined by SET bits of Fig. 3 no matter whether the address is within On-Chip Memory or not. Due to this feature, critical path does not get longer than ordinary cache access. The followings are the procedure of memory access:

1. The corresponding sets decided by SET bits of the address are accessed. In parallel with this access, whether the address is within On-Chip Memory area or not is decided by ASR, AMR and OMV.
2. If the accessed address is within On-Chip Memory, the data from the corresponding way (decided by WAY bits) is selected. Otherwise, ways whose WLR is 0 are accessed as ordinary cache.

## 2.5 Other Architectural Issues

We must consider other architectural issues. One is the guarantee of correct access order of On-Chip Memory and the other is coherence problem between cache and Off-Chip Memory. The former issue implies that execution of page-load/page-store and load/store instructions should wait for the completion of preceding those instructions if the accessed address is the same location on On-Chip Memory. The latter issue implies that if a certain Off-Chip Memory area is accessed by page-load/page-store instructions when cache holds the data of that area, consistency between cache and Off-Chip Memory must be kept. See [6] for the detail description of these issues.

## 2.6 Benefit of On-Chip Memory

The benefit of On-Chip Memory is summarized as follows.

1. *better use of temporal locality*
   Even though temporal locality of data access is extracted, for example by tiling, unwilling line conflicts prevent cache from making good use of the locality. Especially, conflicts between data of plenty locality and those of little locality are very harmful. However, such kinds of conflicts frequently occur in HPC. On-Chip Memory can avoid such interferes because software can do the control explicitly.
2. *suppress of unnecessary data transfer*
   For non-consecutive or stride data access, unnecessary data in the same line is transferred from off-chip memory. This is the waste of valuable off-chip bandwidth and cache space. On-Chip memory avoids this unnecessary data transfer by the block-stride transfer mechanism of page-load and page-store.
3. *improvement of effective bandwidth*
   For consecutive data access, the number of data transfer is reduced by invoking a large amount of data transfer at a time. This is helpful for making effective off-chip bandwidth closer to its theoretical bandwidth. This optimization gets more important when the access latency increases. In ordinary cache, this could be possible by using larger line. However, this also increases the opportunity of line conflicts and unnecessary data transfer. On-Chip Memory realizes a large amount of consecutive data transfer without these sacrifices.

Since off-chip bandwidth will not grow as fast as on-chip bandwidth, reducing off-chip traffic is essentially required. The first and the second benefits contribute to the reduction of off-chip traffic. The third optimization contributes to the relaxation of performance degradation caused by off-chip access latency.

## 3 Optimization of Benchmarks

In this paper, performance of cache-based architecture and SCIMA is evaluated and compared by using two kernels (CG, FT) of NAS Parallel Benchmarks[7] and QCD (Quantum ChromoDynamics) computation[8]. QCD is a practical application used at Center for Computational Physics, University of Tsukuba[9]. In the evaluation, each benchmark is optimized for cache architecture and SCIMA respectively. In this section, we describe the overview of the benchmarks and explain how to optimize them for the two architectures.

Table 2 shows the data sets of target programs used in the evaluation. For saving simulation time, Class-W was selected in CG and FT. On the other hand, the data size of QCD is fairly practical[10]. In QCD, we measured the required time for executing the most time consuming part once. This part is executed so many times in the real computation.

As mentioned later in Section 4.2, we assume the total size of cache/On-Chip is 32KB. This size is quite small but reasonable for evaluating class-W benchmark because cache and On-Chip Memory is usually smaller than the whole data set in HPC.

**Table 2.** Data set of each programs

| program | data set |
|---|---|
| Kernel CG | class W<br>- $p$: 7000 elements (double-precision)<br>- $A$: 7000×7000 (double-precision) |
| Kernel FT | class W<br>- 128×128×32 (double-precision complex) |
| QCD | "G,R,B,V,T": total 2.5MB<br>"U": 1.5MB<br>"M": 3MB |

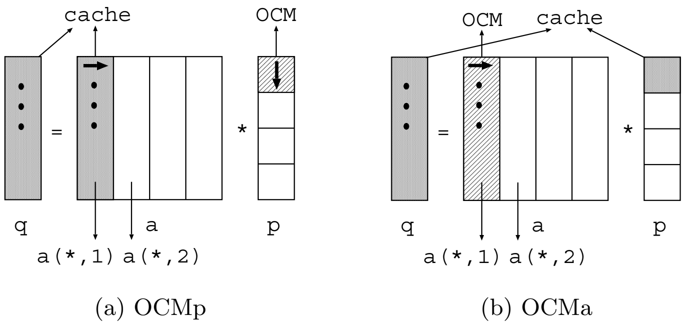


**Fig. 4.** Optimization for SCIMA in Kernel CG

## 3.1 NPB Kernel CG

The time consuming part of CG forms $q = Ap$, where $A$ is sparse matrix and $p$ and $q$ are vectors. The structure of the innermost loop is "$sum = sum + a(k) * p(colidx(k))$". Therefore, $A$ is accessed consecutively whereas $p$ is accessed in random. Another characteristic is that $A$ has no reusability whereas $p$ has reusability.

The original code is optimized for cache-based architecture and SCIMA in several ways as follows.

**Cache-Opt:** To exploit the reusability of $p$, blocking optimization is applied. The computation is blocked in the same way as shown in Fig. 4 except that all the data is of course accessed through cache.

In addition to blocking optimization, the program is optimized for SCIMA in the following two ways.

**SCIMA OCMp:** The vectors $p$ is accessed through On-Chip Memory as shown in Figure 4-[a]. This optimization intends to have the benefit 1 of Section 2.6.

Using this optimization, the reusability of vectors $p$ can be fully utilized without interference within elements of the blocked vector $p$ itself or other arrays.

**SCIMA OCMa:** The sparse matrix $A$ is accessed through On-Chip Memory as shown in Figure 4-[b]. This optimization intends to have the benefit 2 of Section 2.6. High effective bandwidth is expected through this optimization because large amount of data is transferred at once. Through this optimization, it is expected that reusability of vectors $p$ is exploited better because no interference between $p$ and $A$ occurs. This is the benefit 1 of Section 2.6.

### 3.2 NPB Kernel FT

The most time consuming part of this kernel is 3-D FFT. The FFT algorithm has stride data access where the stride size is the power of 2. This leads to frequent line conflicts. Moreover, there are plenty of temporal locality in the core FFT part. Therefore, in the original benchmark code, blocking optimization is applied and each tile is copied into a consecutive temporary area. Although the stride data access still occurs in the part of data copy, frequent line conflicts are avoided during the core FFT calculation. Because blocking optimization has already been applied to the original code, we call the original code **Cache-Opt**.

**SCIMA:** In SCIMA, the temporary area for data copy is allocated on On-Chip Memory. The data copy procedure is realized as stride data transfer from Off-Chip Memory into On-Chip Memory. Thus, the benefit 3 of Section 2.6 is expected. Moreover, reusability in the FFT computation would be exploited better because no interference occurs in On-Chip Memory. This is the benefit 1 of Section 2.6.

### 3.3 QCD Computation

QCD is dynamics governed by quantum field theory, which is a problem of particle physics. In this theory, strongly interacting particles called hadrons are made of fundamental quarks and gluon. Numerical simulation of QCD is formulated on 4-dimensional space-time lattice.

In QCD computation, most of the computation is spent in solving a linear equation. The BiCGStab method, which is an iterative algorithm, is used for solving the linear equation. We analyze the performance of the iteration in BiCGStab.

The iteration consists of RBMULT, LOCALMULT and other routines called inter-MULT. Note that RBMULT routine is the most time consuming part. Table 3 illustrates the computation structure of the iteration. Lattice space is divided into even and odd parts. For example, G_e and G_o reprensent even and odd parts of array G respectively. The second line of Table 3, “B_e(0.5), U(1.5) $\rightarrow$ G_o(0.25)”, for instance, indicates that 0.25MB of array G_o is computed by accessing 0.5MB of array B_e and 1.5MB of array U. Each array has the following characteristics.

- G,R,B,V,T: These arrays have high inter-routine reusability. In addition, they are accessed utmost 8 times in each RBMULT routine.
- U: This array is used only in RBMULT routine which is called 4 times in the iteration. In each RBMULT, U is accessed only once.
- M: This array is accessed only in LOCALMULT routine. Moreover, only even or odd part of M is accessed in one LOCALMULT. Thus, each data of M is accessed only twice in one iteration.

**Table 3.** Iteration Structure in QCD (accessed data size [MB])

| Routine | source | → | destination |
|---|---|---|---|
| inter-MULT 1 | | | |
| RBMULT | B_e(0.5), U(1.5) | → | G_o(0.25) |
| LOCALMULT | G_o(0.25),M_o(1.5) | → | G_o(0.25) |
| RBMULT | G_o(0.5), U(1.5) | → | V_e(0.25) |
| LOCALMULT | V_e(0.25),M_e(1.5) | → | V_e(0.25) |
| inter-MULT 2 | | | |
| RBMULT | R_e(0.5), U(1.5) | → | G_o(0.25) |
| LOCALMULT | G_o(0.25),M_o(1.5) | → | G_o(0.25) |
| RBMULT | G_o(0.5), U(1.5) | → | T_e(0.25) |
| LOCALMULT | T_e(0.25),M_e(1.5) | → | T_e(0.25) |
| inter-MULT 3 | | | |

To summarize these characteristics, while G, R, B, V and T have plenty of reusability, U and M have no intra-routine reusability and a little inter-routine reusability. However, since the iteration is repeated so many times, even U and M are reused over the repeated iterations. We have optimized this computation as follows.

**Cache-Opt:** To exploit reusability of “G,R,B,V,T”, blocking optimization is applied.

**SCIMA:** “U,M” are accessed simultaneously with “G,R,B,V,T” in the iteration loop. Therefore, if only cache is provided, the blocks of “G,R,B,V,T”, which have plenty of reusability, may be flushed out from the cache because of the interferences with “U,M”. To avoid the problem, this code is optimized for SCIMA as follows. “G,R,B,V,T” are still accessed through cache because LRU algorithm would be the best for handling them. On the other hand, “U,M” are accessed through On-Chip Memory to avoid interference with “G,R,B,V,T”. Moreover, high throughput can be obtained by large data transfer size because “U,M” are accessed consecutively to some extent. These optimizations intend to have the benefit 1 and benefit 2 of Section 2.6.

**Table 4.** Assumptions in the Evaluation

| | |
|---|---|
| Execution unit | |
| - integer | 2 |
| - floating-point (multiply-add) | 1 |
| - floating-point (div,sqrt) | 1 |
| - load/store | 1 |
| Cache/On-Chip Memory latency | 2cycle |
| Cache(On-Chip Memory) size | 32KB |
| Cache associativity | 4way |
| Cache line size | 32, 64, 128, or 256B |
| *page* size | 4KB |
| Instruction cache accesses | all hit |
| Branch prediction | perfect |
| Data cache structure | lock-up free L1 cache |
| Execution order | out-of-order |

## 4 Performance Evaluation

### 4.1 Evaluation Environment

SCIMA is defined as an extension of existing architecture. In the evaluation, MIPS IV is selected as the base architecture.

It would be preferable to develop an optimized compiler which can handle the architectural extensions. In the evaluation, however, users specify which data should be located on On-Chip Memory, when those should be transferred between off-chip and on-chip memory, and which location of On-Chip Memory should be used by directives in source programs. Giving directives is not difficult because data accesses are fairly regular. Blocking optimization is also applied by users.

These informations on the usage of On-Chip Memory are specified in source program and compiled by ordinary MIPS compiler. We have developed a preprocessor which inserts these informations into assembly code after the compilation. We have also developed a clock level simulator which accepts the binary object generated by existing compiler and interprets the informations inserted by the preprocessor.

### 4.2 Assumptions for the Evaluation

Table 4 shows the assumptions used in the evaluation. These are common throughout the evaluation.

We assume total on-chip memory (cache and On-Chip Memory) capacity is 32KB. We employ 4-way associative 32KB cache and assume four kinds of line sizes, 32B, 64B, 128B, and 256B.

**Table 5.** Combination of cache and On-Chip Memory

| | cache size (associativity) | On-Chip Memory size |
|---|---|---|
| (a) | 32KB (4way) | 0KB |
| (b) | 24KB (3way) | 8KB |
| (c) | 16KB (2way) | 16KB |
| (d) | 0KB (0way) | 32KB |

By using the reconfiguration mechanism of section 2.4, four combinations of cache and On-Chip Memory are possible as shown in Table 5. We use configuration (a) as a cache-based architecture and configuration (b) or (c) as SCIMA architecture. Here, configuration (d) is not considered at all. because no scalar variable can be cached in this configuration, which obviously leads to poor performance. In the evaluation, configuration (c) is selected in CG and FT, whereas configuration (b) is selected in QCD. The decision on the configuration depends on the property of optimizations and the size of data set.

The assumptions of perfect instruction cache and branch prediction are reasonable because time consuming part of HPC applications consists of regular loops.

### 4.3 Classification of Execution Cycles

The execution time is classified into CPU busy time, latency stall, and throughput stall. Total cycles are obtained under the above assumption. Throughput stall is defined as the cycles which could be saved from total cycles if Off-Chip Memory bandwidth were infinite. Latency stall is defined as the cycles which could be saved further if Off-Chip Memory latency were 0 cycle. The rest is the CPU busy time.

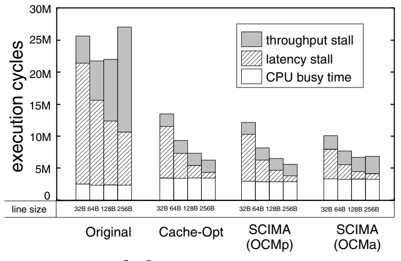
Each time is obtained as follows. First, $C_{normal}$, $C_{th\infty}$, and $C_{perfect}$ are measured by simulation. Here, $C_{normal}$ indicate the cycles under the assumption of Table 4. $C_{th\infty}$ indicate the cycles where Off-Chip Memory bandwidth is infinite. $C_{perfect}$ indicate the cycles where Off-Chip Memory bandwidth is infinite and Off-Chip Memory latency is 0cycle. Then, cycles of each category is calculated as follows.

$$\begin{aligned}\textbf{CPU busy time} &= C_{perfect}\\ \textbf{latency stall} &= C_{th\infty} - C_{perfect}\\ \textbf{throughput stall} &= C_{normal} - C_{th\infty}\end{aligned}$$

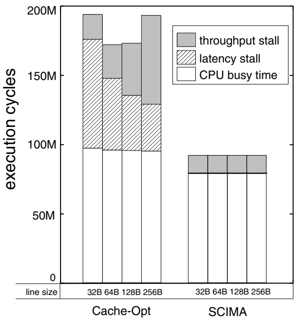
## 5 Evaluation Result

### 5.1 Result

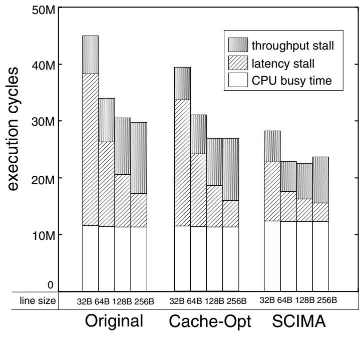
Fig. 5 illustrates the execution cycles and their breakdowns of each program. In this figure, “Original” represents original code which is not modified from the original program. "Cache-Opt" and "SCIMA" represent modified codes optimized for cache and SCIMA architecture respectively. In the case of FT, only "Cache-Opt" and "SCIMA" is given because "Original" code itself have optimizations for cache as mentioned in Section 3.2.



[A] Kernel CG



[B] Kernel FT



[C] QCD

**Fig. 5.** Evaluation Result

**Kernel CG**

The best partitioning (tiled size) is selected through explorative experiments for each optimization. As a result, "Cache-Opt", "OCMp", and "OCMa" are partitioned into 7, 4, and 7 tiles respectively.

"Cache-Opt" achieves about 1.9 times higher performance than "Original" when the line size is 32B. This is because "Cache-Opt" can exploit the reusability of vector $p$, and consequently, latency stall and throughput stall are considerably reduced.

"OCMp" in which $p$ is accessed through On-Chip Memory achieves slightly higher performance than “Cache-Opt”. This illustrates the reusability of $p$ is exploited further on On-Chip Memory. This is brought by the benefit 1 of Section 2.6. However, the improvement is not significant. In the Kernel CG, because access pattern of each array except $p$ is regular and consecutive, cache blocking is quite useful.

“OCMa” in which sparse matrix $A$ is accessed through On-Chip Memory is 1.3 times faster than “Cache-Opt” for 32B cache line. This is because line conflicts between $p$ and $A$ are avoided and because large granularity of data transfer by page-load/page-store reduces latency stall. This is the expectant result of optimization strategy in Section 3.1. However, throughput stall of “OCMa” increases for larger cache line because line conflicts between $p$ and *colidx* increase.

“OCMa” achieves the highest performance for 32B and 64B cache line whereas “OCMp” is the fastest for 128B and 256B cache line. This indicates that the best optimization depends on cache line

**Kernel FT**

“SCIMA” achieves 2.1-1.8 times higher performance than “Cache-Opt”. This superiority is brought by the effectiveness of block-stride transfer feature of page-load/page-store. Due to this feature, latency stall of “SCIMA” is reduced to less than 1% compared with “Cache-Opt”. Throughput stall is also fairly small regardless of cache line size in “SCIMA”. On the contrary, throughput stall of “Cache-Opt” increases extremely for larger cache line size because of unnecessary data transfer. As mentioned in Section 3.2, a tile is copied into a temporary area, In the optimization, the size of each tile was selected as 8KB, which is the capacity of one set within data cache, in order to avoid interferences with other data. Then, the blocked data forms a $4 \times 128$ 2-D array (16B for each data). Thus, when cache line is larger than 64B ($= 4 \times 16$B), unnecessary data transfers occur. SCIMA does not suffer from this problem due to block-stride data transfer.

**QCD**

To compare “Original” with “Cache-Opt”, there is little difference in performance. This illustrates that even if blocking optimization is applied, performance is not improved drastically because of the interferences between “G,R,B,V,T” and “U,M”. However, “SCIMA” in which “U,M” are accessed through On-Chip Memory archives 1.4 times higher performance than “Cache-Opt” and 1.6 times higher performance than “Original” for 32B cache line.

Table 6 shows the traffic of Off-Chip Memory for 32B cache line. The second column “cache” represents the traffic between Off-Chip Memory and cache, whereas the third column “On-Chip Memory” represents the traffic between On-Chip and Off-Chip memories. The last column represents the total Off-Chip Memory traffic. As seen from Table 6, the total traffic of “SCIMA” is about 92% compared with “Cache-Opt”. This indicates that unwilling interferences are avoided by using On-Chip Memory. In this way, our On-Chip Memory contributes to the reduction of Off-Chip Memory traffic.

**Table 6.** Off-Chip Memory Traffic of QCD (32B cache line)

| optimization | cache | On-Chip Memory | total |
|---|---|---|---|
| Original | 26.8MB | 0MB | 26.8MB |
| Cache-Opt | 22.4MB | 0MB | 22.4MB |
| SCIMA | 10.4MB | 11.4MB | 20.8MB |

## 5.2 Discussion

As seen from Fig. 5, for all the benchmarks, latency stall decreases but throughput stall increases for larger line size. This is because more line conflicts are likely to occur for larger line. Therefore, increasing the size of line does not always bring higher performance. Considering the future direction of the semiconductor technology, relative Off-Chip Memory latency is expected to increase and relative Off-Chip Memory bandwidth is expected to decrease. Therefore, it is indispensable to reduce Off-Chip Memory traffic and to make data transfer size larger. Next, we discuss the effectiveness of SCIMA from this viewpoint.

Fig. 6 shows the simulation results of QCD under the assumption representing future semiconductor technology. Performance is evaluated under the three Off-Chip Memory latency (40cycle, 80cycle, 160cycle), and three Off-Chip Memory throughput (4B/cycle, 2B/cycle, 1B/cycle). These assumptions imply future long memory latency and relative narrow bus bandwidth. Fig. 6-[A] is the same as Fig. 5-[C].

From Fig. 6, it is observed that latency stall increases for longer latency and throughput stall increases for narrower bus bandwidth. Consequently, overall performance is greatly degraded as increasing Off-Chip Memory latency and narrowing bus bandwidth. However, the performance of “SCIMA” is less degraded than that of “Cache-Opt”. For example, to compare Fig. 6-[A] (4B/cycle Off-Chip Memory throughput and 40cycle Off-Chip Memory latency) with Fig. 6-[I] (1B/cycle Off-Chip Memory throughput and 160cycle Off-Chip Memory latency) for 128B cache line, performance disparity between “Cache-Opt” and “SCIMA” is widened from 1.2 to 1.4. This result indicates that effectiveness of SCIMA will grow in the future.

## 6 Related Works

There have been many studies on integrating on-chip memory into processor chip besides a cache. Ranganathan et. al.[11] proposed associativity-based partitioning for reconfigurable cache. The mechanism of their cache is similar to our reconfigurable on-chip memory. Chiou et. al.[12] also proposed associativity-based partitioning mechanism which selectively lock some parts of data cache for avoiding unwilling data replacement. However, they do not use their reconfigurable caches as software controllable memory and do not pay much attention

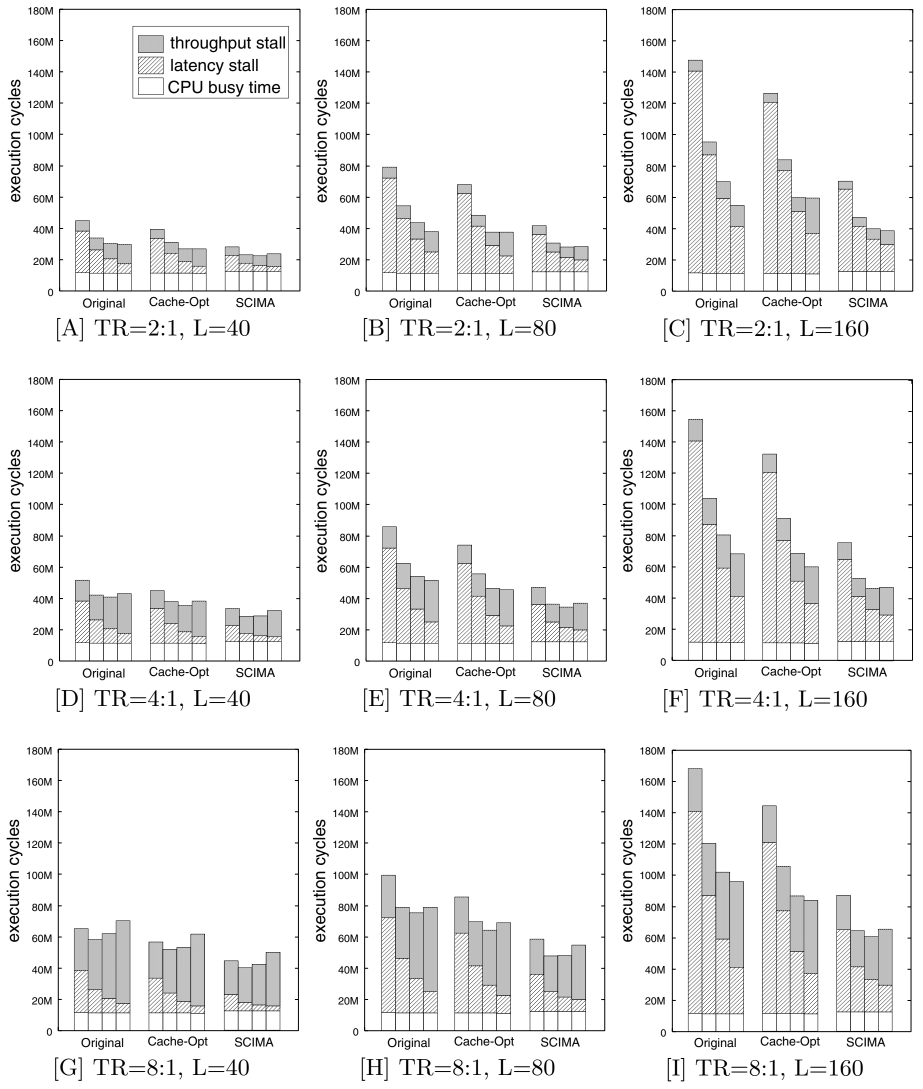


**Fig. 6.** Evaluation Result of QCD under Future Technology ("TR" represents throughput ratio between On-Chip Memory and Off-Chip Memory [On-Chip:Off-Chip] and L represents Off-Chip Memory latency [cycle])

how to supply data from off-chip memory. There is a DSP processor which implements reconfigurable cache[13]. However, it does not focus on the reduction of Off-Chip Memory traffic in HPC. The objective of SCIMA is to make good use of On-Chip Memory by user control and to reduce Off-Chip Memory traffic in HPC.

There have been proposed some processors which adopt small scratch pad RAM [14][15]. However, those scratch pad RAMs are used for specific applications. On the other hand, the target of SCIMA is wide area of HPC applications. Compiler-Controlled Memory[16] is a small on-chip memory which is controlled by compiler. However, this memory is used for only spill code. On-Chip Memory of SCIMA, on the other hand, can be used for all the data if required.

The target of SCIMA is wide area of HPC applications which have large data set. SCIMA realizes flexible and explicit data transfer between on-chip and off-chip memory. This is the main difference between our SCIMA and other works.

## 7 Concluding Remarks

We presented a novel processor architecture *SCIMA* which has software-controllal addressable memory in addition to ordinary cache. The data location and replacement of On-Chip Memory are controlled by software. Due to this feature, SCIMA can control data transfer among memory hierarchy very flexibly. SCIMA has upper compatibility with conventional memory architecture. On-Chip Memory and cache are unified in SCIMA. Therefore, if the whole on-chip memory is used as cache, SCIMA becomes the same as conventional processors.

We presented performance evaluation of SCIMA by using three benchmarks. The evaluation results reveal that SCIMA achieves higher performance than cache-based architecture by reducing both throughput stall and latency stall. The benefit of SCIMA comes from the following features. Firstly, SCIMA can fully exploit temporal locality because unwilling line conflicts are successfully avoided. Secondly, SCIMA can suppress unnecessary data transfer by block-stride data transfer. Finally, SCIMA can improve effective bandwidth by realizing a large amount of consecutive data transfer.

Considering the future direction of the semiconductor technology, off-chip memory latency is expected to increase and relative off-chip memory bandwidth is expected to decrease. Therefore, it is indispensable to reduce off-chip memory traffic and to make data transfer size larger. SCIMA achieves high performance by realizing both issues. This indicates that effectiveness of SCIMA will grow in the future.

From these results, it is concluded that SCIMA is very effective for high performance computing. We are planning to evaluate SCIMA on other wider variety of applications and to design SCIMA in detail for verifying the impact on clock frequency.

## Acknowledgment

We thank the members of the Research Center for Computational Physics, University of Tsukuba for their helpful comments and suggestion. This work is partially supported by the Research for the Future Program of JSPS in the "Computational Science and Engineering" Field (Project No. JSPS-RFTF 97P01102).

## References

1. D. Burger, J. Goodman, and A. Kagi, "Memory Bandwidth Limitation of Future Microprocessor", Proc. 23rd Int'l Symp. on Computer Architecture, pp.78–89, 1996
2. D. Callahan and A. Porterfield, "Data Cache Performance of Supercomputer Applications", Proc. of Supercomputing '91, pp.564–572, 1990.
3. M. Lam, E. Rothberg and M. Wolf, "The cache performance and optimizations of Blocked Algorithms", Proc. ASPLOS-IV, pp.63–74, 1991
4. S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout", Proc. of PLDI'95, pp.279–289, June 1995.
5. P. Panda, H. Nakamura, N. Dutt and A. Nicolau, "Augmenting Loop Tiling with Data Alignment for Improved Cache Performance", IEEE Transactions on Computers, Vol 48, No. 2, pp.142–149, 1999
6. M. Kondo, H. Okawara, H. Nakamura, T. Boku and S. Sakai, "SCIMA: A Novel Processor Architecture for High Performance Computing", Proc. of HPC-Asia 2000, pp.355–360, May 2000
7. D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0", NASA Ames Research Center Report, NAS-05-020, 1995.
8. S. Aoki, R. Burkhalter, K. Kanaya, T. Yoshié, T. Boku, H. Nakamura, Y. Yamashita, "Performance of lattice QCD programs on CP-PACS", Parallel Computing 25, pp.1243–1255, 1999
9. http://www.rccp.tsukuba.ac.jp/
10. M. Kondo, H. Okawara, H.Nakamura, and T. Boku, "SCIMA: Software Controlled Integrated Memory Architecture for High Performance Computing", ICCD-2000, pp.105–111, Oct. 2000
11. P. Ranganathan, S. Adve and N. Jouppi, "Reconfigurable Caches and their Application to Media Processing" Proc. 27th Int'l Symp. on Computer Architecture, pp.214-224, 2000
12. D. Chiou, P. Jain, S. Devadas, and L. Rudolph, "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches", Technical Report CGS-Memo 427, MIT, 1999
13. S. Agarwala, C. Fuoco, T. Anderson, and D. Comisky, "A multi-level memory system architecture for high-performance DSP applications", ICCD-2000, pp.408–413, Oct. 2000
14. Sony's emotionally charged chip. *MICROPROCESSOR REPORT*, Vol. 13, No. 5, 1999.
15. Strongarm speed to triple. *MICROPROCESSOR REPORT*, Vol. 13, No. 6, 1999.
16. K. Cooper and T. Harvey, "Compiler-controlled memory", Proc. of ASPLOS-VIII, pp.2–11, 1998.

# Content-Based Prefetching: Initial Results

Robert Cooksey, Dennis Colarelli, and Dirk Grunwald

University of Colorado
Department of Computer Science
Boulder, CO 80309
{rcooksey, dennisc, grunwald}@cs.colorado.edu

**Abstract.** Memory latency, the time required to retrieve a specified datum from memory, is currently the largest limitation for high-performance computers. Memory prefetching attempts to reduce the memory latency by moving data from memory closer to the processor. Different prefetching mechanisms attempt to model access patterns that may be used by programs. For example, a stride or stream prefetcher assumes that programs will access memory in a linear pattern. In applications that utilize a large number of dynamically allocated objects, the memory access patterns can become very irregular, and difficult to model.
This paper proposes *content-based prefetching*, a method of data prefetching that attempts to overcome the problems introduced by the irregular memory access patterns seen in pointer-intensive applications, thus allowing prefetches of "pointer chasing" references. Content-based prefetching works by examining the content of data as it is moved from memory to the caches. Data values that are likely to be addresses are then translated and pushed to a prefetch buffer. Content-based prefetching has the capability to prefetch sparse data structures, including graphs, lists and trees.
In this paper we examine the issues that are critical to the performance and practicality of content-base prefetching. The potential of the content-aware prediction mechanism is demonstrated and compared to traditional stride prefetching techniques.

## 1 Introduction

### 1.1 Overview

Most prefetch mechanisms work by recording the history of load instruction usage, indexing on either the address or the effective address of the load instruction [6,20]. This requires the prefetcher to have observed the load instruction one or more times before an effective address can be predicted. This method can work well for loads that follow an arithmetic progression, but does not show good performance for pointer loads that may exhibit a more irregular access pattern. An alternative mechanism is to try to find a correlation between miss addresses and some other activity. The correlation [5] and Markov [10] prefetchers record patterns of miss addresses in an attempt to predict future misses, but this technique requires a large correlation table and a training phase for the prefetcher.

Compiler based techniques have been proposed which insert prefetch instructions at sites where pointer dereferences are anticipated. Luk and Mowry [15] showed that taking a greedy approach to pointer prefetching can improve performance despite the increased memory system overhead. Lipasti *et al.* [14] developed heuristics that consider pointers passed as arguments on procedure calls and insert prefetches at the call sites for the data referenced by the pointers. Ozawa *et al.* [19] classify loads whose data address comes from a previous load as *list accesses*, and perform code motions to separate them from the instructions that use the data fetched by list accesses.

Since prefetch mechanisms target different classes of program references, they can be combined to yield a more effective total prefetching behavior; this was explored for the Markov prefetcher [10] and it was found that stride prefetchers improve the performance of the Markov prefetcher by filtering references with arithmetic progressions, leaving more table space for references with different behavior.

In this paper we examine a technique that attempts to predict addresses in pointer-intensive applications using a hardware technique. In general the current hardware-only predictors have two main limitations: they can only predict one instance ahead of the current load address, and they only work with linked-lists. The predictor being proposed here has no built-in biases toward the layout of the recursive data structures being prefetched, and has the potential to run many instances ahead of the load currently being executed by the processor. We will show this is a requirement for pointer-intensive applications, which traditionally do not provide sufficient computational work for masking the prefetch latency. Some hybrid prefetch engines can run several instances ahead of the processor, but they require *a priori* knowledge of the layout of the data structure, and in some cases, the traversal order of the structure.

The prediction strategy being proposed in this paper is based not on the history of load addresses, but on the *content* of the load itself. The *content-based prefetcher* borrows techniques from conservative garbage collection [2]. When data is demand-fetched from memory, each word of the data is examined for a likely address. The memory controller maintains a shadow *translation lookaside buffer* (TLB) that both establishes what values are likely addresses and provides a virtual-to-physical mapping to allow the physical address to be prefetched. The physical addresses are added to a prefetch request queue and then transferred when the memory bus becomes idle.

The remainder of this paper is organized as follows. In Section 2, we present in detail the proposed content-based prefetching mechanism. Section 3 provides a characterization of the workloads that will be used to evaluate the prefetch mechanism. Section 4 presents some of the preliminary results collected from simulation models that were constructed to test the feasibility of content-aware data prefetching. Section 5 presents a survey of both software and hardware prefetching, with a focus on the prior art that is most relevant to the work being presented in this paper. And finally, in Section 6, we discuss our contributions and directions for future work

## 2 Proposed Prefetching Scheme

### 2.1 Brief Overview

The content-based prefetcher works by examining the *content* of loads that either miss in the L1 data cache, or access the main memory. If the content is determined to likely be an address, that address is then loaded from memory into a prioritized prefetch request

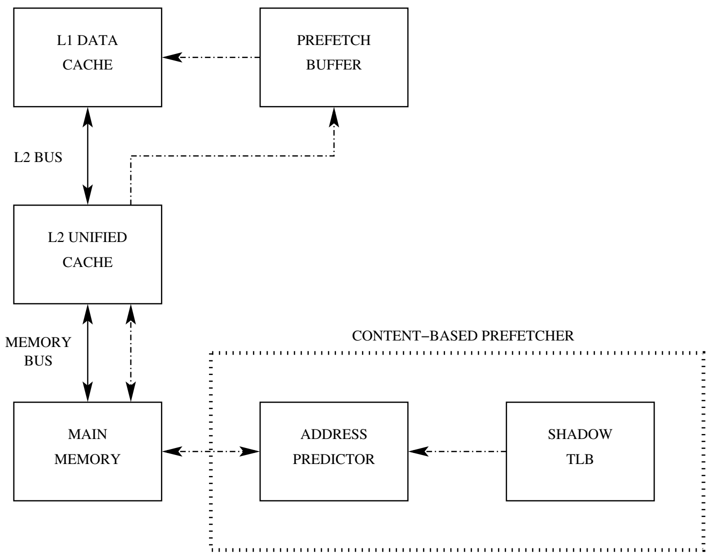


**Fig. 1.** Content-based Prefetcher Architecture.

queue, and when the memory bus is available, the address is pushed up the memory hierarchy chain towards the L1 data cache (see Figure 1). To determine if the content of a load is possibly an effective address, the content-based prefetcher maintains a shadow TLB.

## 2.2 Trigger Mechanism

All data loads first check the L1 data cache. If a load miss occurs, the prefetch buffer is checked. If the load hits in the prefetch buffer, the appropriate cache line is copied into the L1 data cache and the load request is satisfied. To alleviate any coherence issues between the prefetch buffer and the L1 data cache, the prefetch buffer entry is invalidated once the data is moved from the prefetch buffer to the data cache. If a subsequent miss occurs in the prefetch buffer, the miss address is sent to the content-based prefetcher located in the memory controller.

Our initial mechanism simply monitored the load address reference stream seen at main memory (essentially the L2 miss reference stream). This only allowed a limited number of opportunities for the content-based prefetcher to make a prediction. If the prefetcher was successful, this reference stream was reduced even further. For this reason it was decided to also monitor (snoop) the L1 miss address stream.

## 2.3 Prediction Mechanism

The main memory is accessed to get the data located at the effective address of the missed load. If the load is a pointer-load, this data is the effective address of a future load. To determine if the data is such an address, the candidate address is checked using the shadow TLB. If the content of a candidate load results in a shadow TLB hit, the content is deemed to be an address. The main memory is once again accessed to obtain the data for a prefetch located at the newly determined load address. This newly initiated prefetch request is placed into a prioritized prefetch request queue where it waits to be pushed up the memory chain. The content-based prefetcher also incorporates a next-line prefetcher. For every content-determined prefetch request, a prefetch request for the next cache line is automatically placed into the prefetch request queue.

## 2.4 Recursive Scan

Before any prefetch requests are sent towards the L2 cache, the prefetched cache line itself is checked for candidate effective addresses. This recursive check allows the possible detection of next pointers in the case of linked lists, or child pointers in the case of a tree structure. It also provides the same functionality as the look-head program counter of the stride prefetcher: it allows the prefetcher to run ahead of the processor.

The recursive prefetch requests are placed into the prefetch request queue, ordered by the level of recursion at which the likely address was predicted. This results in a breadth-first ordering of the prefetch requests. There is no limit to the number of recursive levels that the prefetcher is allowed to scan. In an effort to keep the prefetch requests as timely and as least-speculative as possible, a primary prefetch request, a request that originated from an L1 load miss (that is, a non-recursive prediction), causes all the outstanding prefetch requests still residing in the prefetch queue to be squashed. The prefetch request queue is scanned before any new requests are added to ensure no duplicate requests reside in the queue.

## 2.5 Data Movement

The movement of data within this model does place new requirements on the memory hierarchy to handle memory requests that were not initiated by the processor. When a memory request is generated by the processor, and a miss occurs, an entry is placed in a *miss status handling register* (MSHR) to record the miss. This provides the cache with the ability to have multiple outstanding misses. When the request has been satisfied and is propagating up the memory chain back towards the CPU, it is guaranteed to find a matching MSHR entry that contains information about the request.

In the model being presented, the prefetch requests are initiated at the memory level, and thus as they move up the chain there is no longer a guarantee that a matching MSHR entry will be found. It is possible to encounter a matching entry if a demand fetch is outstanding for the same cache block. If no matching MSHR entry is found, the request can be stored in a free MSHR entry while waiting to continue its movement towards the processor. If a matching entry is found, the prefetch request will be merged with the demand fetch's MSHR entry, resulting in a partial masking of the demand fetch's load latency. Any demand fetch moving up the chain that fails to find a matching MSHR entry will be dropped, as it will have already been satisfied by a earlier prefetch request. If the address of the prefetch request is currently not residing in the L1 data cache or the prefetch buffer, and no partial masking has occurred, the prefetch request is placed into the prefetch buffer. A prefetch buffer is used for the reasons expected. Prefetched data is speculative, and placing the data directly into the cache could inadvertently displace valid cache lines which are still within the active working set. It also keeps a demand fetch from displacing what would have been a useful prefetch before it could be used.

## 3 Workload Characteristics

### 3.1 Synthetic Benchmark

To evaluate the content-based prefetch mechanism, a *synthetic* benchmark was created that executes long sequences of only pointer-loads. The benchmark does not provide any computational work between loads, and is usable only as a tool for measuring the feasibility of the prefetch mechanism.

This application creates a randomly connected graph of 64-byte nodes. The node size equals the block line size of the various caches, thus reducing any locality benefit the memory system may have gained by having multiple addresses per cache line. Each node points to a single, randomly selected node. The application then repeatedly selects a node, follows that node for 50 pointer traversals, and then selects another starting point to avoid being caught in a loop in the structure. See Figure 2. By using a large array (many times the size of the L1 data cache), and randomly accessing members of the array, this *synthetic* application should produce large cache miss rates.

```
for (int nodes = 0; nodes < REPS; nodes++) {
  void *here = array[nodes];
  void *there;

  for (int refs = 0; refs < 50; refs++ ) {
    there = *here;
    here = there;
  }
}
```

**Fig. 2.** Synthetic benchmark source code.

### 3.2 Olden Benchmark Suite

The Olden benchmarks [23] were originally created to test the Olden C* compiler for the Thinking Machines CM-5. The compiler used software caching and computation migration to improve the performance of programs that used dynamic structures [4]. The initial benchmarks have been stripped of the CM-5 specific code, resulting in sequential code which runs on uniprocessor machines. It has become standard practice to use the uniprocessor version of the Olden benchmarks when performing research on prefetching mechanisms that target pointer-intensive workloads.

The Olden applications are a collection of relatively small applications, each performing a monolithic task. These tasks include sorting, graph optimizations, graphic routines, and scientific code. They manipulate dynamically allocated data structures, which are usually organized in lists or trees, rarely in arrays of structures. For this paper only a subset of the benchmarks were evaluated. This subset includes the majority of data structure types found in the full benchmark suite. A summary of the benchmark set, including a brief description, type of linked data structure used, and input parameters, is provided in Table 1.

**Table 1.** Olden benchmarks.

| *Benchmark* | *Description* | *Data Organization* | *Input* |
|---|---|---|---|
| bisort | Sort of integers using disjoint bitonic sequences | binary-tree | 250000 |
| em3d | Electromagnetic wave propagation in a 3D object | linked lists | 2000 100 75 |
| health | Columbian health care simulation | double-linked lists | 5 500 |
| perimeter | Perimeters of regions in images | quad-tree | 10 |

Table 2 summarizes the dynamic instruction characteristics of each Olden application. The table contains the total number of instructions executed and then provides the percentage of instructions that correspond to memory operations (divided into loads and stores), integer operations, floating point operations, control transfers (including unconditional branches, call/return branches used for subroutines or conditional branches) and the percentage of "miscellaneous" instructions. Also shown is the percentage of branches that are "taken" for each application since this provides some features of the loop-density in each application.

**Table 2.** Olden benchmarks: Instruction mix as a percentage of total instructions executed.

| Program | Instructions | Memory | | Int. | FP. | Control | | | | Misc |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Br | Subr | Cbr | | |
| | | Loads | Stores | Ops | Ops | | | Num | % Taken | |
| bisort | 291,276,993 | 32.00 | 15.83 | 35.98 | 0.00 | 0.81 | 6.30 | 9.06 | 51.85 | 0.00 |
| em3d | 840,666,611 | 31.35 | 5.21 | 44.19 | 7.52 | 0.05 | 1.52 | 10.13 | 70.63 | 0.00 |
| health | 185,371,777 | 37.95 | 12.07 | 30.59 | 0.70 | 0.29 | 2.32 | 16.04 | 89.81 | 0.00 |
| perimeter | 844,715,547 | 15.87 | 9.19 | 52.22 | 0.00 | 2.16 | 9.98 | 10.55 | 29.24 | 0.00 |

As a whole the Olden benchmarks are similar to the *integer* benchmarks found in other benchmark suites (*e.g.* SPEC). They do differ in the area of memory instructions. The percentage of load and store instructions is greater (over 40% in three of the benchmarks), but this memory overhead is to be expected in applications that are allocating and modifying dynamic data structures. The side-affect of the increased memory instruction percentage is a reduction in the integer operation percentage, and a decrease in the amount of computational work available to mask the memory operations.

The L1 data cache miss rates, with no prefetching (referred to as the *base* case in the paper), are given in Table 3. These miss rates were measured using an 8KB, 2-way set associative cache with 64 byte cache blocks. This is the standard L1 data cache configuration used throughout this paper. Surprisingly both *bisort* and *perimeter* have very small miss rates. These small *a priori* miss rates do not provide a lot of opportunity for a prefetcher to improve the memory system performance of these applications.

**Table 3.** Olden benchmark L1 data cache miss rates.

| Benchmark | Loads Executed | L1 Data Cache Misses | L1 Data Cache Miss % |
|---|---|---|---|
| bisort | 93,217,887 | 3,198,387 | 3.43 |
| em3d | 263,623,159 | 38,762,235 | 14.70 |
| health | 70,347,560 | 18,836,688 | 26.78 |
| perimeter | 126,128,928 | 5,603,401 | 4.44 |

The following subsections provide an examination of each of the Olden benchmarks being used in this paper. This includes a brief description of the task being performed, a discussion of the data structures used, how they are accessed, and an evaluation of whether the content-based prefetch mechanism presented in this proposal will provide a performance improvement. ATOM [28] was used to locate those segments of code that were the largest contributors to the application's poor cache performance.

**Bisort.** The main data structure in *bisort* is a binary tree. The tree is generated at the start of the program, and is populated with a set of randomly generated integers. The program then performs both a forward and backward sort of this data. The sorting algorithm creates a bitonic sequence in each subtree, and then merges the subtrees to obtain the sorted result. The definition of a tree node is given in Figure 3.

```
struct node {
    int value;
    struct node *left;
    struct node *right;
}
```

**Fig. 3.** Bisort binary-tree node definition.

As indicated earlier, *bisort* exhibits fairly good cache performance without assistance from prefetchers. Table 3 shows that the L1 cache miss rate is less than 3.5%. This does not provide many opportunities for a prefetcher to improve the cache performance. This low miss rate can be attributed somewhat to the binary-tree's layout in memory. The tree is generated in traversal order, with each data cache line able to hold two tree nodes. This leads to the program benefiting from both spatial and temporal locality. The spatial locality can interfere with the content-based prefetchers ability to issue prefetches. Unlike a stride prefetcher that has the benefit of examining the entire L1 memory reference stream, the content-based prefetcher only sees the L1 miss reference stream. The reduced L1 cache miss rate due to data locality reduces the miss reference stream, which reduces the opportunities provided to the content-based prefetcher.

An examination of *bisort* shows that over 90% of the cache misses occur in Bimerge(), which contains both loop control structures and recursive calls. A more detailed examination shows the majority of the misses (over 88%) can be attributed to a specific set of loads - those loads that reference the right child of a given node. This is seen both with and without prefetching. This cache miss behavior leads to a discussion of the overall difficulty of prefetching nodes of a tree-based data structure. While the content-based prefetcher is not intentionally biased against tree-based structures, the current implementation of the prefetcher shows poor performance for applications that use tree-based data structures. This is not really a fault of the prefetcher, but an indication of how difficult it is to issue timely and useful prefetches for such structures. The prefetcher will examine each tree node, find the left and right pointer, and issue a prefetch request for each. During a normal in-order tree traversal, the dominate traversal path used in *bisort*, the left child pointers will be followed until a leaf node is encountered. During this initial depth-first traversal, the prefetcher will be issuing prefetches for both the left and right child nodes as it encounters each node on the initial path to the leaf node. For a tree with a large number of levels, such as found in *bisort*, the right child prefetches, while potentially useful, are being initiated well in advance of their use. They will most likely be replaced in the prefetch buffer before having the opportunity to be useful. In practice this is seen by the high data cache miss rates when a right child pointer is followed.

The current breadth-first approach of the content-based prefetcher will prove beneficial when a right child node reference closely follows a left child reference, which most likely occurs near the leaf nodes. Here the right child prefetches will have a higher probability of residing in the prefetch buffer and eliminating a potential cache miss. Thus for *bisort*, the predominate cache miss will still be right child pointer references, with the content-based prefetcher providing a modest reduction in the overall L1 data cache miss count.

**EM3D.** *EM3D* models the propagation of electromagnetic waves in a 3D object. This object is represented as a bipartite graph containing both E nodes and H nodes. At each time step of the simulation, the E node values are updated using the weighted sum of the neighboring H nodes. This process is repeated for the H nodes. The main computation loop consists of walking down a list of nodes, obtaining the values of neighboring nodes, and using those values to update the current node.

Over 95% of the L1 data cache misses occur in the function compute_nodes() (see Figure 4). While the outer loop is a classic example of the pointer-chasing problem, the inner loop which contains the list traversals is not. The list traversals are actually array traversals, with the individual node addresses being calculated, not loaded. For the inner loop where the bulk of the misses result from array and scalar references, content-based prefetching should yield little improvement. The prefetcher should detect the pointer-chasing outer loop, but the improvement will be minimal as the length of the traversals performed in the loop are small.

```
void compute_nodes(node_t *nodelist) {
   int i;
      for(; nodelist; nodelist = nodelist->next);
         for(i=0; i < nodelist->from_count; i++);
            node_t *other_node = nodelist->from_nodes[i];
            double coeff = nodelist->coeffs[i];
            double value = other_node->values;
      }
   }
}
```

**Fig. 4.** Function compute_nodes() from the em3d benchmark.

**Health.** *Health* simulates the Columbian health care system. Using the terminology from [25], *health* utilizes a "backbone-and-ribs" structure. The "backbone" is a four-way tree, with the "ribs" being doubly linked lists. Each node of the tree represents a hospital, and at each of these nodes is a linked-list of patients. At each time step of the simulation, the tree is traversed, and the patients are evaluated. Upon evaluation, the patients are either treated, or they are passed up the tree to the parent node.

Close to 90% of the L1 data cache misses are involved in the linked-list traversals in the two functions addList() and
check_patients_waiting(). Very few misses result from the traversal of the quad-tree. This is not surprising as the quad-tree is actually rather small, is created in traversal order, and remains unchanged during the execution of the program. The patient lists are modified frequently, and thus are doubly linked (contain both a back and forward pointer) to make the insertion and deletion of patients (nodes) easier. The linked-list node definitions are given in Figure 5. The list traversals are performed in a pointer-chasing manner, and should benefit significantly from the content-based prefetcher. But, the *List* node structure may limit this benefit somewhat. By being doubly linked, the *back* pointer will cause prefetches to be issued for the previous node visited, a prefetch for a data item that is obviously still resident in the cache. This will create memory system overhead that can not provide any cache performance benefit. These *back* pointers will also lead the prefetcher to recursively issue prefetches in reverse traversal order. This further increases the memory overhead dedicated to prefetches which will be of no benefit, and may affect the timeliness of useful prefetches.

```
struct Patient {
    int hosps_visited;
    int time;
    int time_left;
    struct Village *home_village;
}
```

(a) Patient data structure

```
struct List {
    struct Patient *patient;
    struct List *back;
    struct List *forward;
}
```

(b) List data structure

**Fig. 5.** Health data structure definitions.

Also a problem is the ordering of the pointers in the *List* node structure. The *back* pointer is placed ahead of the *forward* pointer. During the prefetcher's recursive scan stage, the prefetch request initiated as a result of detecting the *back* pointer will be issued before the *forward* initiated prefetch request. This "backward" prefetch is for the node just visited during the traversal, which is a wasted prefetch for a node that is already in the cache. Further, it occupies memory system bandwidth, delaying the "forward" prefetch. A simple experiment was performed to see if such pointer ordering would affect the performance of the content-based prefetcher. From Table 3 (page 39), we see the L1 data cache base miss rate for *health* is 26.78%. Quite high, which should provide ample opportunity for the content-based prefetcher to improve the memory system performance. Using the *back* - *forward* pointer ordering shown in Figure 5, the prefetcher was able to reduce the L1 data cache miss rate to 12.05%. A significant improvement. Swapping the *back* - *forward* pointer ordering resulted in a L1 data cache miss rate of 11.58%. This would indicate that the content-based prefetcher is sensitive to the pointer ordering, an area where compiler support could be beneficial.

**Perimeter.** The program *perimeter* computes the perimeter of a set of quad-tree encoded raster images. The encoded raster image is generated at the start of the program. As the individual nodes of the tree are allocated, each node is assigned one of three colors: white, black, or grey. In practice only two of these colors are assigned. The leaf nodes of the tree are designated as being black, and all non-leaf nodes are tagged as being grey. Once this raster image (quad-tree) is generated it is never modified, and is only traversed in a predetermined order. The definition of a tree node is given in Figure 6.

Two observations should be made at this point. First, *perimeter* already exhibits good cache performance without prefetching. The L1 cache miss rate is just under 4.5%.

```
typedef struct quad_struct {
    Color color;
    ChildType childtype;
    struct quad_struct *nw;
    struct quad_struct *ne;
    struct quad_struct *sw;
    struct quad_struct *se;
    struct quad_struct *parent;
}
```

**Fig. 6.** Perimeter quad-tree node definition.

So like *bisort*, minimal opportunity is provided for the prefetcher to increase the cache performance. Second, the quad-tree is generated in traversal order which results in the tree residing contiguously in memory in traversal order. From a stride prefetcher perspective, this effectively changes the memory access pattern to that of an array traversal.

The quad-tree's initial traversal during allocation should give a stride prefetcher the needed training to start predicting addresses. During the tree traversal when the perimeter is calculated, the stride prefetcher will be seeing the memory access reference pattern for the second time, allowing prefetch requests to be generated. Thus a stride prefetcher should be able to lower the L1 data cache miss rate.

The content prefetcher may not be able to provide such an increase. *Perimeter*, like *bisort*, highlights some of the problems seen when trying to prefetch nodes within a tree-based data structure. The main problem is the quad-tree structure (see Figure 6). Each node of the tree contains five pointers: four child node pointers and a parent node pointer. Each node of the tree requires 48 bytes, which the memory allocator expands to 64 bytes. The result is the tree node byte count equals the cache line byte count of the cache simulators used in this paper. The recursive scan feature of the content-based prefetcher (see Section 2.4, page 36) will issue prefetch requests for all five pointers, including the parent pointer. The tree is traversed in an in-order fashion: nw, ne, sw, and se. While the prefetcher is taking a more breadth-first approach to prefetching, the tree traversal is more characteristic of a depth-first traversal. This results in prefetch requests being generated long before they would possibly be used. With a fixed-sized prefetch buffer, these untimely but useful prefetches will most likely get replaced prior to being beneficial. The high branching factor of the quad-tree (versus a binary tree) only makes the problem more pronounced. The parent node prefetch requests cause a separate problem in that they will occupy memory system overhead prefetching data (nodes) that already resides in the cache. So the combination of a small *a priori* L1 data cache miss rate, the quad-tree structure, and the constant in-order traversal of the tree structure, the content-based prefetcher will most likely only provide a small performance improvement.

### 3.3 Summary

A perusal of the related work section will show that the Olden benchmark suite has become the standard set of applications to be used when evaluating a prefetch mechanism that attacks the pointer-chasing problem. These benchmarks do allocate and access a large number of dynamic data structures, but as discussed in several earlier sections, the majority of these structures are immutable, and are accessed in an array-like manner. So while all of them are indeed pointer-intensive applications, not all them exhibit what would be considered "traditional" pointer-intensive characteristics.

## 4 Preliminary Results

### 4.1 Metrics

Traditionally *coverage* and *accuracy* have been used to measure the "goodness" of a prefetch mechanism. While the authors of [27] argue that such metrics may be deceiving as they do not provide a direct indication of performance, they are sufficient for purposes of this paper. In this paper we will be using the coverage and accuracy definitions provided in [27].

Prefetch coverage is a measure of the cache performance gains that the prefetcher provides, and is defined as the percentage of cache misses eliminated by the prefetcher. This is shown in Equation(4).

$$coverage \ = \ prefetch\ hits\ /\ misses\ without\ prefetching \tag{1}$$

Prefetch accuracy estimates the quality of the prefetching algorithm, and is defined as the percentage of prefetch requests that were useful (they matched a future load). This is shown in Equation(5).

$$accuray \ = \ useful\ prefetches\ /\ number\ of\ prefetches\ generated \tag{2}$$

A third metric that is used in this paper is *miss rates*. While miss rates are not the best metric (they do not reflect the timeliness of the prefetches), for the simple memory model which is *not* a cycle accurate model, miss rates can be used as a measure of the prefetcher's performance.

### 4.2 Simple Memory Model

For the initial investigation of content-based prefetching, a simple two-level cache simulation tool was developed using ATOM [28]. This cache hierarchy contains separate L1 data and instruction caches, and a unified L2 cache. TLBs are maintained for both the instruction and data L1 caches, with a shadow TLB assisting with the content-based prefetch address prediction. The L1 caches are 8KB 2-way set associative with 64 byte lines; the unified L2 cache is a 1MB 4-way set associative with 64 byte lines. We explicitly use small caches because the workloads tend to have a small working set size. The parameters for the various components of the memory model are shown in Table 4.

The timing for the memory model is simple, and does not model any contention points (*e.g.* busses). Instructions and memory references complete in one cycle. When

**Table 4.** Simple memory model cache simulation parameters.

| Caches | | | |
|---|---|---|---|
| Component | Capacity | Block Size | Assoc. |
| IL1 | 8KB | 64 bytes | 2-way set |
| DL1 | 8KB | 64 bytes | 2-way set |
| UL2 | 1MB | 64 bytes | 4-way set |

| TLBs | | | |
|---|---|---|---|
| Component | Entries | Page Size | Assoc. |
| ITLB | 64 | 8KB | fully |
| DTLB | 64 | 8KB | fully |
| STLB | 1024 | 8KB | direct |

prefetching is enabled, a queue is used to store the outstanding prefetch request. Each cycle the first request in the prefetch request queue is moved to the prefetch buffer, providing a prefetch request throttling mechanism. When content-based prefetching is enabled, recursive content scans are performed prior to moving a request from the prefetch request queue to the prefetch buffer. The memory model also contains a stride prefetcher, but because of the simplistic timing model, it does not utilize a look-ahead program counter. Stride generated prefetch requests are also queued and use the same throttling mechanism, the difference being the stride prefetcher does not perform recursive cache line scans. It should be noted that this simulator does not include a processor model, but instead uses a straight in-order fetch-execute cycle.

### 4.3 Synthetic Benchmark

The *synthetic* benchmark by design should benefit greatly from the content-based prefetcher. While the data structure is defined as a random graph, the individual path traversals through the graph can be viewed as traversals of random linked-lists. With the simplified memory model allowing prefetches to be available one cycle after being issued, the content-based prefetcher should be able to eliminate all the base case (no prefetching) L1 data cache misses.

The prefetch coverage and accuracy measurements for the content-based prefetcher when executing the *synthetic* benchmark are shown in Table 5. The prefetch coverage is quite high, almost 97%, showing that the prefetcher is eliminating the majority of the L1 data cache misses that occur when no prefetching is used. The result of this high coverage can be seen in Table 6, where the L1 data cache miss rate drops from the base miss rate of over 66%, to just over 2%.

**Table 5.** Content-based prefetcher coverage and accuracy measurements when executing the synthetic benchmark.

| Benchmark | Load Misses w/o PF | Load Misses with PF | Predictions Made | Good Predictions | Coverage % | Accuracy % |
|---|---|---|---|---|---|---|
| synthetic | 37,238,406 | 1,152,641 | 315,104,010 | 36,085,765 | 96.90 | 11.45 |

**Table 6.** L1 data cache miss rates when executing the synthetic benchmark.

| | Loads | Baseline | | Stride | | Content | |
|---|---|---|---|---|---|---|---|
| Benchmark | Executed | Misses | Miss % | Misses | Miss % | Misses | Miss % |
| synthetic | 56,032,887 | 37,238,406 | 66.46 | 36,928,423 | 65.90 | 1,152,641 | 2.06 |

Remember that the *synthetic* benchmark ends each path after traversing 50 nodes, starting a new path at a random node in the graph. This graph is many times larger than the L1 data cache, so there is a large probability that the new path's starting node will result in an L1 data cache miss. These misses can not be eliminated by the prefetcher, and rely solely on locality for cache hits. Thus every 50th node reference, or 2% of the node references, have a high probability of incurring a cache miss. The L1 data cache miss rate with the content-based prefetcher enabled is 2%. This implies that the content-based prefetcher is eliminating all the possible cache misses along each path, with the exception of the first node in each path.

While the content-based prefetcher achieves a high coverage percentage, it's accuracy is quite low at 11.45%. This indicates that a large majority of the prefetches initiated by the content-based prefetcher are not useful. This is somewhat expected for several reasons. First, the content-based prefetcher also incorporates a next-line prefetcher. For every address that is predicted, and a prefetch request generated, the same is done for the next cache line following the predicted address. For the *synthetic* benchmark where each node in the graph occupies a complete cache line, and the paths are random, next-line prefetching provides no benefit. So at least half of the prefetch requests will not be useful. Second, unlike a stride prefetcher which is able to perform a tag lookup in the L1 data cache and prefetch buffer prior to initiating a new prefetch, the content-based prefetcher will be initiating prefetches for cache lines that are already resident in either the L1 data cache or the prefetch buffer. So while these prefetch requests are correct, they are not useful, and have needlessly occupied memory bandwidth. This highlights the need for a feedback mechanism to help in filtering prefetch requests.

## 4.4 Olden Benchmarks

In Section 3, the workload analysis of the Olden benchmarks concluded that the only benchmark that could benefit from using the content-based prefetcher would be *health*. The results from running the benchmarks using the simple memory model proves this to be true. A look at Tables 7 and 8 show that the L1 data cache miss rate for *health* is significantly reduced, achieving a prefetch coverage of 55%. The remaining three benchmarks have prefetch coverage ranging from 10% to 23%. The tree-based applications, *bisort* and *perimeter* benefit from temporal locality of the recursive calls near the tree leaf nodes when traversing the trees. The large branching factor of *perimeter* keeps it from achieving the same prefetch coverage of *bisort*. While most of the linked-list accesses in *em3d* are performed in an array-like fashion, *em3d* does benefit from the content-based prefetching of future instances of nodes in the "backbone" tree data structure.

While *perimeter* is defined as a pointer-intensive application, the stride prefetcher performed considerably better than the content-based prefetcher. This is an obvious indication of regularity in the memory reference stream. The workload analysis of *perimeter*

**Table 7.** Olden benchmark coverage and accuracy.

| Benchmark | Load Misses w/o PF | Load Misses with PF | Predictions Made | Good Predictions | Coverage % | Accuracy % |
|---|---|---|---|---|---|---|
| bisort | 3,198,387 | 2,460,219 | 556,145,424 | 5,259,635 | 23.08 | 0.95 |
| em3d | 38,762,235 | 31,668,436 | 2,104,177,098 | 24,370,120 | 18.30 | 1.16 |
| health | 18,836,688 | 8,476,011 | 647,730,592 | 10,391,124 | 55.00 | 1.60 |
| perimeter | 5,603,401 | 5,013,004 | 688,412,386 | 5,603,401 | 10.53 | 0.82 |

**Table 8.** Olden benchmark L1 data cache miss rates.

| Benchmark | Loads Executed | Baseline | | Stride | | Content | |
|---|---|---|---|---|---|---|---|
| | | Misses | Miss % | Misses | Miss % | Misses | Miss % |
| bisort | 93,217,887 | 3,198,387 | 3.43 | 2,761,909 | 2.96 | 2,460,219 | 2.64 |
| em3d | 263,623,159 | 38,762,235 | 14.70 | 30,621,473 | 11.62 | 31,668,436 | 12.01 |
| health | 70,347,560 | 18,836,688 | 26.78 | 18,170,315 | 25.83 | 8,476,011 | 12.05 |
| perimeter | 126,128,928 | 5,603,401 | 4.44 | 1,981,680 | 1.57 | 5,013,004 | 3.97 |

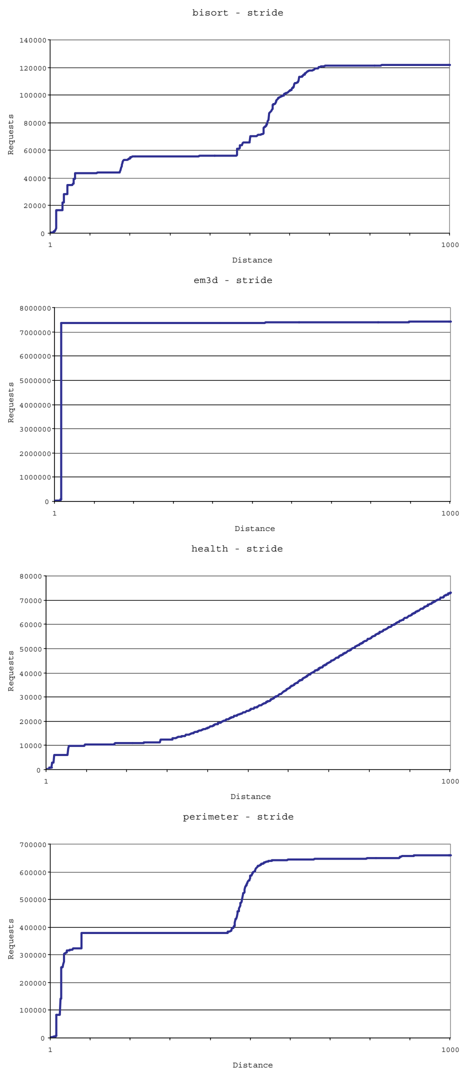
showed that the tree structure used in *perimeter* was contiguous in memory in traversal order and remained unchanged for the duration of the program's execution, characteristics that a stride prefetcher should be able to exploit. Table 8 shows the stride prefetcher was indeed able to take advantage of this regularity, attaining a prefetch coverage of over 64%.

Of concern are the extremely low prefetch accuracies. The numbers indicate that the content-based prefetcher is issuing 100 "bad" prefetch requests for each "good" request. This is not a problem in the simplified timing model, but in a cycle-accurate memory model that includes resource contention, the large memory system overhead introduced by these useless prefetches will have to impact the timeliness of the useful prefetch requests. These low prefetch accuracies further highlight the need for a filtering mechanism to remove as many of the useless prefetches as possible.

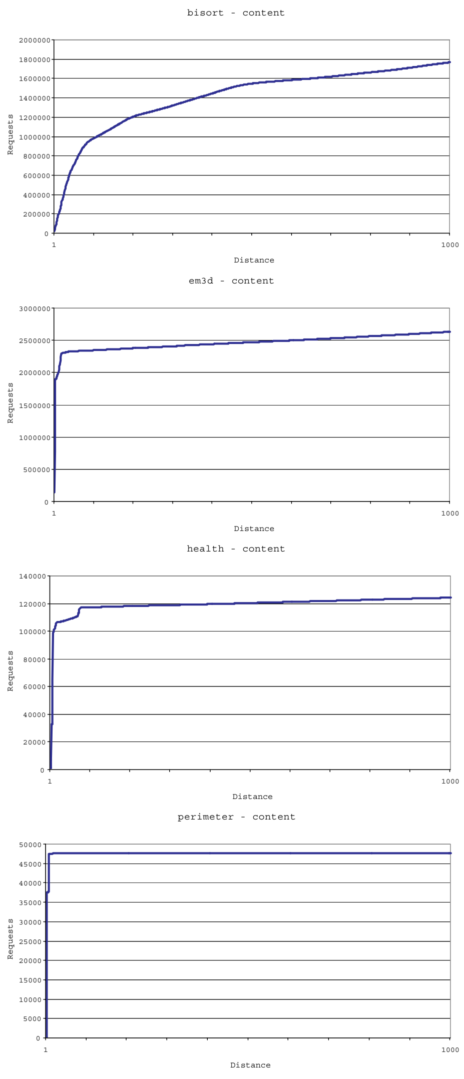
### 4.5 Prefetch Distances

Although the timing model used in this initial investigation is simplistic, it is interesting to examine the prefetch distance (the time from when a prefetch is issued to when the item is used), shown for both stride and content-based prefetchers in Figure 7 . Stride prefetchers tend to have a long prefetch distance while the content prefetcher tends to have a shorter prefetch distance. This implies that very little computational work is available to mask the content-based prefetches. Intuitively this makes sense as most programs immediately use a pointer once it has been loaded. The problems that such small distances introduce is a continuous theme through the relevant prior art. An observation is that the content-based prefetcher must be allowed to run ahead of the current program counter if is to improve the memory system performance

The differences in prediction-usage distances between stride and content-based prefetching imply that each prefetcher is making different predictions. This would indicate that there is an opportunity to combine both prefetchers for better overall prediction performance.



(a) stride



(b) content

**Fig. 7.** Address Prediction-Usage Distances. Each graph displays a cumulative histogram of the distance, measured in instructions, between when a prefetch request is generated, and the prefetched datum is used.

## 5 Related Work

A large volume of work has been generated in the areas of software [3] and hardware prefetching, including both data prefetching [7] and instruction prefetching [9]. The following subsections present a survey of software, hardware, and hybrid data prefetching mechanisms. While the survey is biased towards prefetchers that target pointer intensive applications, it is not limited to only those mechanism that attack the pointer-load problem.

### 5.1 Software Based Prefetching

While the following subsections describe several software-based mechanisms for implementing data prefetching, it should be noted that software-controlled prefetching does require support from the hardware. The *instruction set architecture* (ISA) for the processor must provide a *prefetch* instruction. The software uses this instruction to issue a data request to the memory subsystem. The caches must be lock-up free [13], which allows the cache memory to have multiple outstanding misses. The program can then continue executing while the memory system retrieves the specified datum, assuming the application does not need the requested data.

*Sequential Predictors.* Mowry *et al.* [18] presented a general method of hiding the latency of a load by scheduling a matching speculative non-faulting prefetch load in advance of the demand load. Their algorithm is selective in that it identifies those load references that are likely to result in a cache miss, and inserts prefetch instructions only for them. The outstanding prefetch requests were queued in a prefetch issue buffer. This algorithm is intended for applications that operate on dense matrices.

*Recursive Data Structures.* Luk and Mowry [15,16] proposed and evaluated three compiler algorithms for scheduling prefetches for *recursive data structures* (RDS): greedy prefetching, history-pointer prefetching, and data-linearization prefetching. Of these three algorithms, only the greedy algorithm was implemented in the SUIF optimizing research compiler. History-pointer prefetching and data-linearization prefetching were evaluated using hand optimized code.

The greedy algorithm uses type information to detect which objects belong to a recursive data structure, and control structure information to recognize when these objects are being traversed. Once an RDS object has been found, and a likely control structure that contains an RDS traversal has been located, the greedy algorithm will insert prefetches of all pointers within the object that point to other nodes of the RDS. This insertion will be made at the the earliest point where these addresses are available within the surrounding control structure.

History-pointer prefetching creates new jump-pointers (history-pointers) within an RDS. These history-pointers contain observed values of recent traversals of the RDS. The history-pointers are constructed using a FIFO queue which holds pointers to the last *n* nodes that have been visited, where *n* is set to equal the prefetch distance. The history-pointer for the oldest node in the queue is set to the current node.

Data-linearization prefetching attempts to map heap-allocated nodes that are likely to be accessed close together (such as traversal order in the case of a binary-tree) into contiguous memory locations. The benefit is that the RDS can now be seen as an array-like structure, allowing next node pointers to be computed instead of dereferenced. This method only works well for RDSs that are fairly constant once created, as the process of dynamically remapping the RDS may result in large runtime overheads.

*SPAID.* Lipasti *et al.* [14] proposed the SPAID (Speculatively Prefetching Anticipated Interprocedural Dereferences) heuristic, a compiler-based prefetching scheme for pointer-intensive and call-intensive applications. SPAID is based on the premise that procedures are likely to dereference any pointer passed to them as arguments. SPAID therefore inserts prefetches for the objects pointed to by these pointer arguments at the call sites. This method defines the prefetch distance to be the start of procedure and the dereference of the passed pointer. Thus the prefetches are only effective if this distance is comparable to the cache load miss latency.

### 5.2 Hardware Based Prefetching

*Stride Prefetcher.* Stride-based prefetchers take advantage of regularities in the data memory reference stream. In [6], Chen and Baer proposed a hardware prefetcher that monitors the stride between successive memory references of load instructions. The *reference prediction table* (RPT), a cache whose entries are tagged with the instruction address of load instructions, detects strides by keeping a history of distances between subsequent load effective addresses. To allow the RPT to issue prefetch requests in a timely manner, a *look-ahead program counter* (LAPC) is utilized to engage the prefetcher in advance of the processor's regular program counter.

*Tango.* Pinter and Yoaz [21] introduce Tango, a prefetching mechanism that leverages the effective utilization of slack time and hardware resources not being used by the main computation. The Tango prefetcher is comprised of four hardware components. The *program progress graph* (PPG) is a directed graph where nodes correspond to a branch instruction and edges to a block of instructions on the path between the two corresponding branch instructions. A *reference prediction table for superscalar processors* (SRPT) stores the access history information of the memory reference instructions. The *pre-PC* and PC are set equal at the start of the program and following any mispredicted branches. Using the information in the PPG, the pre-PC is capable of advancing to the next block in one cycle. The *prefetch request controller* (PRC) controls the scheduling of the data prefetch requests. A unique feature of the PRC is a queue which contains previous data cache accesses that hit. By examining this queue, the PRC is able to detect redundant prefetches without consuming any of the data cache's tag-port bandwidth.

*Stream Prefetcher.* Jouppi [11] introduced *stream buffers* as a method to improve the performance of direct-mapped caches. Stream buffers prefetch sequential streams of cache lines, doing this independently of the program context. Stream buffers are implemented as FIFO buffers that prefetch succeeding cache lines, starting with a missed cache line.

The stream buffers presented in [11] do have a limitation in that they can only detect streams which contain unit strides.

Palacharla and Kessler [20] addressed this limitation by extended the stream buffer mechanism to detect non-unit strides, along with introducing a noise rejection scheme to improve the accuracy of the stream buffers. This filtering mechanism is simple in that it will wait for two consecutive misses to a sequential cache line address before allocating a stream buffer.

Farkas *et al.* [8] enhanced stream buffers by augmenting them with an associative lookup capability and a mechanism for detecting and eliminating the allocation of stream buffers to duplicate streams.

Sherwood *et al.* [26] introduced *predictor-directed stream buffers*, which improved the stream buffer's performance in pointer-base applications, using a stride-filtered Markov address predictor to guide the stream buffer prefetching.

*Correlation-Based Prefetchers.* The first instance of correlation-based prefetching being applied to data prefetching is presented in a patent application by Pomerene and Puzak [22]. A hardware cache is used to hold the parent-child information. A further innovation they introduce is to incorporate other information into the parent key. They suggest the use of bits from the instruction causing the miss, and also bits from the last data address referenced. They also introduce a confirmation mechanism that only activates new pairs when data that would have been prefetched would also have been used. This mechanism is very much like the *allocation filters* introduced by Palacharla *et.al.* [20] to improve the accuracy of stream buffers and serves a similar purpose here.

Charney and Reeves [5] extended the Pomerene and Puzak mechanism and applied it to the L1 miss reference stream rather than directly to the load/store stream. They improved upon the previous mechanism by introducing a FIFO history buffer which allowed a greater lead time for the prefetches. Instead of entering parent-child pairs into the pair cache, ancestors older than the parent can be paired with the child and entered in the pair cache. Another important contribution by Charney and Reeves was to show that stride based prefetching could be combined with correlation based prefetching to provide significant improvements in prefetch coverage over using either approach alone, on certain benchmarks.

Alexander and Kedem [1] proposed a mechanism similar to correlation-based prefetching but used a *distributed* prediction table. In their variation, a correlation-based table was used to predict bit-line accesses in an Enhanced DRAM, and was used to prefetch individual bit lines from the DRAM to the SRAM array.

Joseph and Grunwald [10] use a *Markov model* to predict future memory references. The model is implemented in hardware as a prediction table, and allows multiple, prioritized prefetch requests to be launched. The prefetcher is designed to act as an interface between the on-chip and off-chip cache, and can be added to most existing computer designs.

*Recurrence Recognition.* Mehrotra and Harrison [17] contribute a memory access classification scheme that represents address sequences as recurrent patterns. They then exploit this scheme to extend the RPT described in [6] to allow it to capture memory reference patterns associated with recursive data structures.

*Dependence-Based.* Roth *et al.* [24] proposed a *dependence-based prefetching* mechanism that dynamically captures the traversal behavior of *linked data structures* (LDS). The dependence-based prefetcher works by matching producer-consumer instruction pairs. A producer being a load whose value is an address, and a consumer being a load that uses that value (address) as its base address. The prefetch mechanism uses three hardware components. The *potential producer window* (PPW) is a list of the most recent loaded values and the corresponding load instruction. Producer-consumer pairs are recorded in the *correlations table* (CT). The collection of producer-consumer pairs residing in the CT define the *traversal kernel* for the LDS. Prefetch requests are enqueued onto a *prefetch request queue* (PRQ). The PRQ buffers prefetch requests until data ports are available to service them. One drawback to this approach is that is does limit prefetching to a single instance ahead of a given load.

*Jump-Pointers.* In [25], Roth and Sohi investigate the use of jump-pointers [15] to prefetch LDSs that contain *backbone* and *backbone-and-ribs* structures. A backbone LDS contains only one type of node, and is connected in a recursive manner (*e.g.* list, tree, graph). A backbone-and-ribs LDS contains secondary structures at each of the primary structure nodes (*e.g.* a linked-list at each node of a tree).

Roth and Sohi combined two prefetching techniques, jump-pointer prefetching and chained prefetching (using only the original pointers found in the structure), to form four prefetching idioms: queue, full, chain, and root jumping. Queue jumping is applied to backbone structures, and adds jump-pointers to each node of the structure. Full jumping adds only jump-pointers to backbone-and-ribs structures. Chain jumping adds jump-pointer prefetches to the backbone, and chained prefetches to the ribs. Root jumping uses only chained prefetches.

## 5.3 Hybrid Prefetching

*Push Model.* Yang and Lebeck [29] describe a *push* model of generating prefetch requests for linked data structures (LDS). In most prefetch mechanisms, the prefetch request originates at the upper, or CPU level of the memory hierarchy. The request then propagates down the memory chain until it can be satisfied, and then traverses back up to the level where the prefetch request was initiated. In the push model, prefetch engines are attached to each level of the memory hierarchy and “push” prefetched data towards the upper levels. This eliminates the request from the upper levels to the lower level, which can dramatically reduce the memory latency of the prefetch request. This should allow for the overlapping of data transfers from node-to-node within a linked data structure.

The prefetch engines are very similar to the prefetch mechanism described in [24]. They are designed to execute traversal kernels which are down-loaded to the prefetch engine via a memory-mapped interface. The root address of the LDSs are conveyed to the prefetch engines using a special “flavored” load instruction. Upon seeing such a load, the prefetch engines begin executing the previously downloaded traversal kernel independent of the program execution.

*Memory-Controller Based.* Zhang *et al.* [30] present a memory-controller based prefetching mechanism, where data prefetching of linked data structures originate at the memory controller. The mechanism they propose is implemented using the Impulse Adaptable Memory Controller system. The memory controller is under the control of the operating system, which provides an interface for the application to specify optimizations for particular data structures. The programmer, or compiler, inserts directives into the program code to configure the memory controller.

*Prefetch Arrays.* Karlsson *et al.* [12] extend the jump-pointer prefetching discussed in [15,25] by including arrays of jump-pointers, a *prefetch array*, at the start of recursive structure nodes. The premise is to aggressively prefetch all possible nodes of a LDS several iterations prior to their use. To ameliorate the overhead of issuing blocks of prefetches, the ISA is extended to include a *block prefetch operation*.

## 6 Conclusions and Future Work

The current improvements with content-based prefetching are promising. We are experimenting with a number of prioritization and control mechanisms to increase the prefetch accuracy. Content-based prefetching is intended to be used in conjunction with other prefetching mechanisms, such as stride or stream prefetching. This paper explored each prefetcher in isolation, but the difference in the prefetch distance clearly shows that each has unique advantages.

Once we better understand how content-based prefetchers behave, we intend to examine this prefetching mechanism in more depth using a cycle-accurate timing model.

## References

1. T. Alexander and G. Kedem. Distributed predictive cache design for high performance memory system. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, San Jose, California, February 1996. IEEE Computer Society TCCA.
2. H-J. Boehm. Hardware and operating system support for conservative garbage collection. In *IWMM*, pages 61–67, Palo Alto, California, Oct 1991. IEEE Press.
3. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.
4. M.C. Carlisle and A. Rogers. Software caching and computation migration in olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38. ACM, July 1995.
5. M.J. Charney and A.P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.
6. T-F. Chen and J-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Boston, MA, October 1992. ACM.
7. T-F. Chen and J-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, Chicago, IL, April 1994. ACM.

8. K.I. Farkas and N.P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, Chicago, IL, April 1994. ACM.
9. W-C. Hsu and J.E. Smith. A performance study of instruction cache prefetching methods. *IEEE Transactions on Computers*, 47(5):497–508, May 1998.
10. D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, Denver, CO, June 1997. ACM.
11. N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 388–397. ACM, 1990.
12. M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 206–217, Toulouse, France, January 2000. IEEE Computer Society TCCA.
13. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87. ACM, 1981.
14. M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R. R. Roediger. Spaid: Software prefetching in pointer and call intensive enviornments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 231–236, Ann Arbor, MI, November 1995. ACM.
15. C-K. Luk and T.C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996. ACM.
16. C-K. Luk and T.C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999.
17. S. Mehrotra and L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 133–140, Philadelphia, PA USA, May 1996. ACM.
18. T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, October 1992. ACM.
19. T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics an preloading techniques for general-purpose programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 243–248, Ann Arbor, MI, November 1995. ACM.
20. S. Palacharla and R.E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, April 1994. ACM.
21. S.S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 214–225, Paris, France, December 1996. ACM.
22. J. Pomerene and et.al. Prefetching system for a cache having a second directory for sequentially accessed blocks. Technical Report 4807110, U.S. Patent Office, Feb 1989.
23. A. Roger, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.
24. A. Roth, A. Moshovos, and G.S. Sohi. Dependance based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, San Jose, CA, October 1998. ACM.

25. A. Roth and G.S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, Atlanta, GA, May 1999. ACM.
26. T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, Monterey, CA USA, December 2000. ACM.
27. V. Srinivasan, E. Davidson, and G. Tyson. A prefetch taxonomy. Technical Report CSE-TR-424-00, University of Michigan, April 2000.
28. A. Srivastava and A. Eustace. Atom a system for building customized program analysis tools. In *PLDI'94*, pages 196–205, Orlando, FL, June 1994.
29. C-L. Yang and A.R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 176–186, Santa Fe, NM USA, May 2000. ACM.
30. L. Zhang, S.A. McKee, W.C. Hsieh, and J.B. Carter. Pointer-based prefecthing within the Impulse adaptable memory controller: Initial results. In *Solving the Memeory Wall Problem Workshop (ISCA 27)*, June 2000.

# Memory System Support for Dynamic Cache Line Assembly

Lixin Zhang, Venkata K. Pingali, Bharat Chandramouli, and John B. Carter

School of Computing
University of Utah
Salt Lake City, UT 84112
{lizhang, kmohan, bharat, retrac}@cs.utah.edu
http://www.cs.utah.edu/impulse/

**Abstract.** The effectiveness of cache-based memory hierarchies depends on the presence of spatial and temporal locality in applications. Memory accesses of many important applications have predictable behavior but poor locality. As a result, the performance of these applications suffers from the increasing gap between processor and memory performance. In this paper, we describe a novel mechanism provided by the Impulse memory controller called *Dynamic Cache Line Assembly* that can be used by applications to improve memory performance. This mechanism allows applications to gather on-the-fly data spread through memory into contiguous cache lines, which creates spatial data locality where none exists naturally. We have used dynamic cache line assembly to optimize a random access loop and an implementation of Fast Fourier Transform (FFTW). Detailed simulation results show that the use of dynamic cache line assembly improves the performance of these benchmarks by up to a factor of 3.2 and 1.4, respectively.

## 1 Introduction

The performance gap between processors and memory is widening at a rapid rate. Processor clock rates have been increasing 60% per year, while DRAM latencies have been decreasing only 7% per year. Computer architects have developed a variety of mechanisms to bridge this performance gap including out-of-order execution, non-blocking multi-level caches, speculative loads, prefetching, cache-conscious data/computation transformation, moving computation to DRAM chips, and memory request reordering. Many of these mechanisms achieve remarkable success for some applications, but none are particularly effective for irregular applications with poor spatial or temporal locality. For example, no

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

systems based on conventional microprocessors can handle the following loop efficiently if the array A is sufficiently large:

```
float A[SIZE];
for (i = 0; i < itcount; i++) {
   sum += A[random()%SIZE];
}
```

We are developing a memory system called Impulse that lets applications control how, when, and what data is placed in the processor cache [2]. We do this by adding an optional extra level of *physical-to-physical* address translation at the main memory controller (MMC). This extra level of translation enables optimizations such as "gathering" sparse data into dense cache lines, no-copy page coloring, and no-copy superpage creation. In this paper, we describe a new mechanism called *dynamic cache line assembly* that we are considering for Impulse. This mechanism allows applications to request that a cache line be loaded on-the-fly with data from disjoint parts of memory. Applications that can determine the addresses that they will access in the near future can request that data from those addresses be fetched from memory. This mechanism lets applications create spatial locality where none exists naturally and works in situations where prefetching would fail due to bandwidth constraints. Simulation indicates that dynamic cache line assembly improves the performance of the random access loop above by a factor of 3.2 and the performance of the dominant phase of FFTW by a factor of 2.6 to 3.4.

The rest of the paper is organized as follows. Section 3 briefly describes the basic technology of Impulse. Section 4 presents the design details of dynamic cache line assembly. Section 5 studies the performance evaluation of the proposed mechanism. And finally, Section 6 discusses future work and concludes this paper.

## 2 Related Work

Much work has been done to increase the spatial and temporal locality of regular applications using static analysis. Compiler techniques such as loop transformations [1] and data transformations [3] have been useful in improving memory locality of applications. However, these methods are not well suited to tackle the locality problem in irregular applications where the locality characteristics are not known at compile time.

A hybrid hardware/software approach to improving locality proposed by Yamada et al. [12] involves memory hierarchy and instruction set changes to support combined data relocation and prefetching into the L1 cache. Their solution uses a separate relocation buffer to translate array elements' virtual addresses into the virtual relocation buffer space. The compiler inserts code to initiate the remapping, and it replaces the original array references with corresponding relocation buffer references. However, this approach can only relocate strided array references. Also, it saves no bus bandwidth because it performs relocation at the processor. Contention for cache and TLB ports could be greatly increased because the collecting procedure of each relocated cache line must access the cache and CPU/MMU multiple times. This approach is also not designed for irregular applications.

There has been some work in developing dynamic techniques for improving locality. Ding and Kennedy [5] introduce the notion of dynamic data packing, which is a run time optimization that groups data accessed at close intervals in the program into the same cache line. This optimization is efficient only if the gathered data is accessed many times to amortize the overhead of packing and if the access order does not change frequently during execution. DCA setup incurs much lesser overhead because it does not involve data copying, and it allows frequent changes to the indirection vector.

To the best of our knowledge, hardware support for general-purpose cache line gathering such as is supported by DCA is not present in any architecture other than Cray vector machines. For example, the Cray T3E [10] provides special support for single-word load. Sparse data can be gathered into contiguous E-registers and the resulting blocks of E-registers can then be loaded "broadside" into the processor in cache line sized blocks, thus substantially reducing unnecessary bus bandwidth that would have been used in normal cache line fills. Dynamic cache line assembly provides similar scatter gather capability for conventional microprocessors without the need for special vector registers, vector memory operations in the instruction set, or an SRAM main memory.

## 3 Impulse Architecture

The Impulse adaptable memory system expands the traditional virtual memory hierarchy by adding address translation hardware to the main memory controller (MMC) [2, 11, 14]. Impulse uses physical addresses unused in conventional systems as remapped aliases of real physical addresses. For instance, in a system with 32-bit physical addresses and one gigabyte of installed DRAM, the physical addresses inside [0x40000000 – 0xFFFFFFFF] normally would be considered invalid. [1] Those, otherwise unused, physical addresses refer to a *shadow address space.*

Figure 1 shows how addresses are mapped in an Impulse system. The real physical address space is directly backed up by physical memory; its size is exactly the size of installed physical memory. The shadow address space does not directly point to any real physical memory (thus the term *shadow*) and must be remapped to real physical addresses through the Impulse MMC. How the MMC interprets shadow addresses presented to it is configured by the operating system.

This virtualization of unused physical addresses can provide different views of data stored in physical memory to programs. For example, it can create cache-

[1] It is common to have I/O devices mapped to special "high" addresses. This problem can be easily avoided by not letting shadow address space overlap with I/O devices addresses.

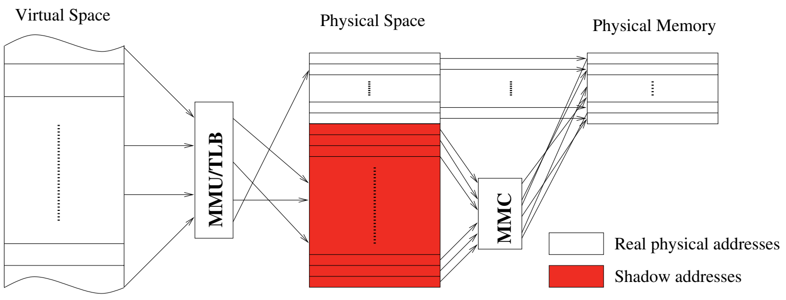


**Fig. 1.** Address mapping in an Impulse system.

friendly data structures to improve the efficiency of the processor caches. The operating system manages all of the resources in the expanded memory hierarchy and provides an interface for the application to specify optimizations for particular data structures. The programmer (or the compiler) inserts appropriate system calls into the application code to configure the memory controller.

To map a data item in the shadow address space to the physical memory, the Impulse MMC must first recover its virtual address. To avoid directly handling virtual addresses at the MMC, we require that the virtual address must be located inside a special virtual region. The OS creates a dense, flat page table in contiguous physical addresses for the special virtual region. We call the page table the *memory controller page table*. The OS then pins down this page table in main memory and sends its starting physical address to the memory controller so that the MMC can access this page table without interrupting the OS. Since data items in a shadow region are mapped to a special virtual region, the MMC only need compute offsets relative to the starting address of the virtual region. We call such an offset a *pseudo-virtual address*. For each shadow data item, the MMC first computes its pseudo-virtual address, then uses the memory controller page table to determine the data item's real physical address. To speed up the translation from pseudo-virtual to physical addresses, the MMC uses an TLB to store recently used translations. We call this TLB the *MTLB*.

Figure 2 shows a simplified block diagram of the Impulse memory system. The critical component of the Impulse MMC is the shadow engine, which processes all shadow accesses. The shadow engine contains a small *scatter/gather SRAM buffer* used as a place to scatter/gather cache lines in the shadow address space, some *control registers* to store remapping configuration information, an ALU unit *(AddrCalc)* to translate shadow addresses to pseudo-virtual addresses, and a *Memory Controller Translation Lookaside Buffer* (MTLB) to cache recently used translations from pseudo-virtual addresses to physical addresses. The control registers are split into eight different sets and are capable of saving configuration information for eight different mappings. However, all mappings share the same ALU unit and the same MTLB.

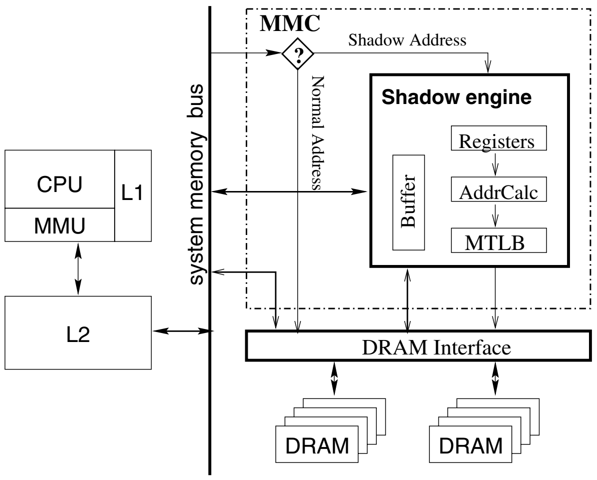


**Fig. 2.** Impulse architecture.

## 4 Design

The proposed dynamic cache line assembly mechanism is an extension of Impulse's *scatter/gather through an indirection vector* remapping mechanism. Its main goal is to enable applications to access data spread through memory as if it were stored sequentially. In this section, we first talk about scatter/gather through an indirection vector, then describe the design of dynamic cache line assembly.

### 4.1 Scatter/Gather through An Indirection Vector

The Impulse system supports a remapping called *scatter/gather through an indirection vector*. For simplicity, we refer to it as *IV remapping* throughout the rest of this paper. IV remapping maps a region of shadow addresses to a data structure such that a shadow address at offset *soffset* in the shadow region is mapped to data item addressed by *vector[soffset]* in the physical memory.

Figure 3 shows an example of using IV remapping on a sparse matrix-vector product algorithm. In this example, *Pi* is an alias array in the shadow address space. An element *Pi[j]* of this array is mapped to element *P[ColIdx[j]]* in the physical memory by the Impulse memory controller. For a shadow cache line containing elements *Pi[j]*, *Pi[j+1]*, ..., *Pi[j+k]*, the MMC fetches elements *P[ColIdx[j]]*, *P[ColIdx[j+1]]*, ..., *P[ColIdx[j+k]]* one by one from the physical memory and packs them into a dense cache line.

Figure 4 illustrates the gathering procedure. The shadow engine contains a one cache line SRAM buffer to store indirection vectors. We call this SRAM buffer the *IV buffer*. When the MMC receives a request for a cache line of *Pi[]*, it loads the corresponding cache line of *ColIdx[]* into the IV buffer, if the IV buffer

```
for (i = 0; i < n; i++) {
   sum = 0;
   for (j = Rows[i]; j < Rows[i+1]; j++)
       sum += Data[j] * P[ColIdx[j]];
   b[i] = sum;
}
```

**Original code**

```
Pi = AMS_remap(P, ColIdx, n, ...);
for (i = 0; i < n; i++) {
   sum = 0;
   for (j = Rows[i]; j < Rows[i+1]; j++)
       sum += Data[j] *  Pi[j];
   b[i] = sum;
}
```

**After remapping**

**Fig. 3.** *Scatter/gather through an indirection vector* changes indirect accesses to sequential accesses.

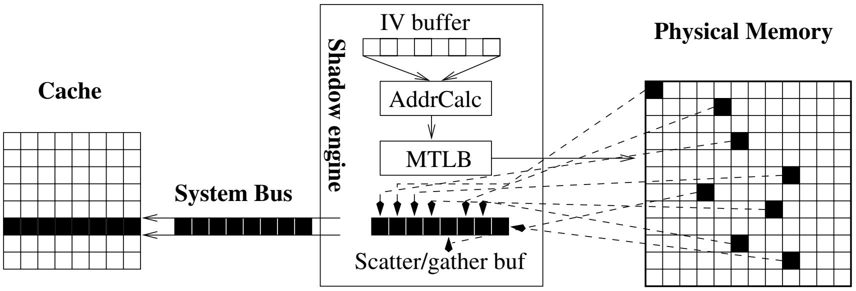


**Fig. 4.** Visualize the gathering procedure through an indirection vector.

does not already contain it. The MMC then can interpret one element of the indirection vector per cycle. The indirection vector may store virtual addresses, array indices, or even real physical addresses. What it stores (addresses or indices) is specified when the remapping is configured; without loss of generality, we will refer to the contents of the IV buffer generically as “addresses” throughout the rest of this paper. If the IV buffer stores virtual address or array indices, the MMC passes each entry to the AddrCalc unit to generate a pseudo-virtual address and translates the pseudo-virtual address to a physical address using the MTLB. Once the MMC has a physical address for a data element, it uses this address to access physical memory. When a data item returns, it is packed into a dense cache line in the scatter/gather buffer.

By mapping sparse, indirectly addressed data items into packed cache lines, *scatter/gather through an indirection vector* enables applications to replace indirect accesses with sequential accesses. As a result, applications reduce their bus bandwidth consumption, the cache footprint of their data, and the number of memory loads they must issue.

A naive implementation of IV remapping requires that the indirection vector exists in the program and its number of elements be the same as the number of data items being gathered, e.g., `Rows[n]` in Figure 3. We extend IV remapping to implement *dynamic cache line assembly*, which can be used by programs where *no indirection vector exists naturally* and where the size of an indirection vector need not be equal to the number of data items being gathered.

## 4.2 Dynamic Cache Line Assembly

The basic idea of *dynamic cache line assembly* is to create indirection vectors dynamically during program execution and to access them using Impulse's IV remapping mechanism. The indirection vectors typically are small, usually smaller than a page. We choose small indirection vectors because accessing them and the resulting small alias arrays leaves a very small footprint in the cache. The small indirection vectors and alias arrays can be reused to remap large data structures.

To use DCA, the application performs a system call to allocate two special ranges of shadow addresses and have the operating system map new virtual addresses to these two ranges. The first range is used to store the addresses from which the application wishes to load data (we call this the *address region*), while the second range is used to store the requested data (we call this the *data region*). The number of addresses that can be stored in the address region is the same as the number of data items that can be stored in the data region. There is a one-to-one mapping between elements of the two ranges: the $i^{th}$ element of the address region is the address of the $i^{th}$ element of the data region. The operating system also allocates a contiguous real physical memory to back up the address region in case an indirection vector is forced out of the IV buffer before the MMC has finished using it.

After setting up the regions, the operating system informs the MMC of their location, as well as the size of each address and the size of the object that needs to be loaded from each address. For simplicity, we currently require that both the address and data regions are a multiple of a cache line size, and that the data objects are a power of two bytes.

After setup, the application can exploit dynamic cache line assembly by filling a cache line in the address region with a set of addresses and writing it back to memory through a cache flush operation. When the MMC sees and recognizes this write-back, it stores the write-back cache line into both the IV buffer and the memory. Storing the write-back into the memory is necessary because the IV buffer may be used by another writeback. When the MMC receives a load request for a cache line in the data region, it checks to see if the IV buffer contains the corresponding cache line in the address region. If the required cache line is not in the IV buffer, the shadow engine loads it from memory. The MMC then interprets the contents of the IV buffer as a set of addresses, passes these addresses through the AddrCalc unit and the MTLB to generate the corresponding physical addresses, fetches data from these physical addresses, and stores the fetched data densely into the scatter/gather buffer inside the shadow engine. After an entire cache line has been packed, the MMC supplies it to the system bus from the scatter/gather buffer.

Figure 5 shows how dynamic cache line assembly can be used to improve the performance of the random access loop presented in Section 1. In this example, we assume that L2 cache lines are 128 bytes, so each cache line can hold 32

```
float *aliasarray;
int   *idxvector;
/* aliasarray[i] <== A[idxvector[i]] */
setup_call(A, SIZE, 32, &aliasarray, &idxvector);
for (i = 0; i < itcount/32; i++) {
   for (k = 0; k < 32; k++)
      idxvector[k] = & (A[random()%SIZE]);
   flush_cache line(idxvector);
   memory_barrier();
   for (k = 0; k < 32; k++)
      sum += aliasarray[k];
   purge_cache line(aliasarray);
}
```

**Fig. 5.** Using *dynamic cache line assembly* on the random access loop

addresses or 32 floats[2]. The program first allocates a 32-element address region `idxvector` and a 32-element data region `aliasarray` through the system call `setup_call()`. In each iteration, the application fills a cache line's worth of the address region with addresses, flushes it, and then reads from the corresponding shadow data region to access the data. Traditional microprocessors with out-of-order execution usually give reads higher priority than writes, so the read request for the data may be issued before the flush occurs. To ensure this does not happen, we insert a *memory barrier* after the flush. Since the same data region is used in every iteration, the old data in the cache must be invalidated so that the next fetch will go to the MMC to retrieve the right data.

The main overhead of using dynamic cache line assembly is that the program must handle the address regions (i.e., indirection vectors) and the MMC must gather a cache line using multiple DRAM fetches. In this example, filling a cache line of the indirection vector introduces 32 sequential memory accesses that do not exist in the original code. Fortunately, those 32 accesses generate only one cache miss. Accessing a cache line of the data region results in another cache miss. As a result, the Impulse version has 2 cache misses for every 32 data accesses. In the original code, however, the same 32 data accesses generate roughly `(32 * (1 - sizeof(cache) / sizeof(A))` cache misses when the array `A` is larger than the cache. Dynamically gathering a cache line in the MMC is much more expensive than fetching a single dense region of memory. Consequently, the code in Figure 5 will be memory-bound because there is a long waiting time in each

[2] It is not required that the objects being fetched are the same size as an address. If, for example, you want to dynamically fetch quad-precision floating point numbers (16 bytes), each time the application flushes a line of addresses, the MMC will fetch four cache lines full of data.

iteration for the MMC packing and returning data. However, its performance can be improved using unroll-and-jam.

```
#define PrecomputeAddresses(start, end)             \
         for (k = start; k < end; k++)              \
            idxvector[k] = &(A[random()%SIZE]);     \
         flush_cache line(&(idxvector[start]));      \
         memory_barrier();                          \
         prefetch_cache line(&(aliasarray[start]));

#define AccessData(start, end)                      \
         for (l = start; l < end; l++)              \
            sum += aliasarray[l];                   \
         purge_cache line(&(aliasarray[start]));

float *aliasarray;
int   *idxvector;
/* aliasarray[i] <== A[idxvector[i]] */
setup_call(A, SIZE, 64, &aliasarray, &idxvector);
PrecomputeAddresses(0, 32);
for (i = 0; i < itcount/64 - 1; i++) {
   PrecomputeAddresses(32, 64);
   AccessData(0, 32);
   PrecomputeAddresses(0, 32);
   AccessData(32, 64);
}
......
```

**Fig. 6.** Using unroll-and-jam with dynamic cache line assembly.

Figure 6 illustrates how unroll-and-jam can be used to improve performance. By using two cache lines for each of the address and data regions, we can overlap computation on one line's worth of data while prefetching the next line, thereby hiding the long memory latency of dynamic cache line gathering. To support this optimization, we need to increase the size of the IV buffer to two cache lines. With software unroll-and-jam, the processor may flush back one cache line of the address region while the MMC is gathering data from another set of address. With two cache lines in the IV buffer, the second write-back can be saved in the buffer instead of being written back to DRAM and then reloaded when needed.

## 5 Performance Evaluation

We evaluated the performance of dynamic cache line assembly using execution-driven simulation. We compared the performance of two benchmarks using dynamic cache line assembly with the same benchmarks unmodified. The first benchmark is the synthetic "random walk" microbenchmark described in Section 1. The second benchmark is a three-dimensional FFT [6] program from DIS benchmark suite [7]. Both benchmarks were compiled using the SPARC SC4.2 compiler with the *-xO4* option to produce optimized code.

### 5.1 Simulation Environment

Our studies use the execution-driven simulator URSIM [13] derived from RSIM [9 URSIM models a microprocessor close to MIPS R1000 [8] and a split-transaction MIPS R10000 cluster bus with a snoopy coherence protocol. It also simulates the Impulse adaptable memory system in great detail. The processor is a four-way, out-of-order superscalar with a 64-entry instruction window. The D/I unified TLB is single-cycle, fully associative, software-managed, and has 128 entries. The instruction cache is assumed to be perfect. The 64-kilobyte L1 data cache is non-blocking, write-back, virtually indexed, physically tagged, direct-mapped, and has 32-byte lines and one-cycle latency. The 512-kilobyte L2 data cache is non-blocking, write-back, physically indexed, physically tagged, two-way associative, and has 128-byte lines and eight-cycle latency. The split-transaction bus multiplexes addresses and data, is eight bytes wide, has a three-cycle arbitration delay and a one-cycle turn-around time. The system bus, memory controller, and DRAMs have the same clock rate, which is one third of the CPU clock. The memory supports critical word first. It returns the critical quad-word for a load request 16 bus cycles after the corresponding L2 cache miss occurs. The memory system contains 8 banks, pairs of which share an eight-byte wide bus between DRAM and the MMC.

The address translation procedure in the shadow engine is fully pipelined. For each shadow access entering the pipeline, the engine generates the first physical address four cycles later and one physical address per cycle afterwards, provided that no MTLB miss occurs. On an MTLB miss, the pipeline is stalled until the required page table entry has been loaded into the MTLB. The MTLB is configured to be four-way associative, with 256 entries and a one-memory-cycle lookup latency.

### 5.2 Results

The performance results presented here are obtained through complete simulation of the benchmarks, including both kernel and application time, the overhead of setting up and using dynamic cache line assembly, and the resulting effects on the memory system.

| | Elapsed cycles | TLB hit rate | L1 hit rate | L2 hit rate | Miss rate | Memory latency | Speedup |
|---|---|---|---|---|---|---|---|
| Base | 196M | 73.67% | 65.68% | 7.76% | 26.56% | 51 cycles | |
| Impulse | 61M | 99.97% | 93.37% | 6.21% | 0.42% | 113 cycles | 3.22 |

**Table 1.** Performance results for the microbenchmark.

**Microbenchmark** In the synthetic microbenchmark, `A[]` holds one million elements and two million random accesses are performed. In theory, its cache hit rate should be the size of the cache divided by the size of `A[]`. So larger `A[]` has smaller cache hit rate and likely yields better performance improvement with dynamic cache line assembly. We choose `A[]` to contain one million elements simply because it is large enough to prove the effectiveness of dynamic cache line assembly and its simulation can complete in a reasonable amount of time.

Table 1 presents the results of this experiment. The unrolled version of the microbenchmark shown in Figure 6, which uses DCA, executes 3.2 times faster than the baseline version. The Impulse version increases the L1 cache hit rate from 65.68% to 93.37% and reduces the number of accesses that are handled by the main memory from 26.56% to 0.42%. One nice side effect of dynamic cache line assembly is the improved TLB performance. The base version of this benchmark has very bad TLB behavior because the TLB is not big enough to hold the translations for the entire array `A[]`. After using dynamic cache line assembly, the TLB needs at most two entries to hold the translations for the address and data regions. Table 1 shows that the TLB hit rate has indeed been greatly improved (from 73.67% to 99.97%).

Average memory latency increases from 51 cycles to 113 cycles, because dynamic cache line assembly requires more work than a simple dense cache line fill, but the improved cache performance overwhelms the effect of the increased memory latency. The memory latency reported here is the average latency for all load accesses, excluding prefetch accesses. The average memory latency of prefetch accesses reaches around 600 cycles due to high MTLB miss rate (82.97%). The MTLB is configured to be four-way set associative with 256 entries. The simulated system uses four-kilobyte base page, so the MTLB's maximum reach is only one megabyte, much less than the `A[]`, which is four megabytes. In the real hardware we are building, the MTLB has 1024 entries. We reduced the MTLB size to 256 entries in our simulations to generate high MTLB miss rates, while leaving the MTLB larger than the CPU TLB (an important feature of Impulse [11]). The good performance of dynamic cache line assembly even with such high MTLB miss rates gives us confidence that our results will hold, and perhaps even improve, on larger data structures. Despite the high latency of dynamic cache line assembly, the use of prefetching results in an average latency of demand requests of 113 cycles. In this microbenchmark, not enough work is done on each piece of data for prefetching to completely hide the load latency, so we still see a high average memory latency. If more work were performed per data item, the memory latency perceived by the processor would drop.

**FFT** Fast Fourier Transform(FFT) is generally characterized by poor temporal and spatial locality. FFTW [6] is a specific implementation of FFT whose self-optimizing approach lets it outperform most other FFT implementations. We chose this FFT implementation as our baseline and modified it to use dynamic cache line assembly.

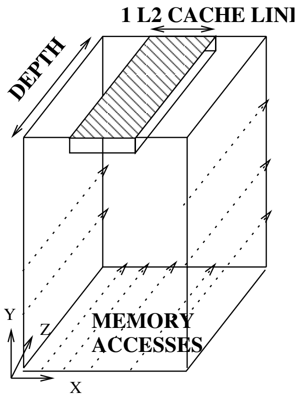


**Fig. 7.** Shows memory access pattern of depth phase and a *cache-column*

In general, 3D FFTW operates in two phases. The 3D input array is accessed along $x$ and $y$ axes in the first phase of the computation. In the second phase, which we call the *Depth Phase*, data is accessed along the $z$ axis. For large arrays, row-major array layout causes poor locality when the array is accessed along the $y$ and $z$ dimensions. The memory performance accesses along the $y$ dimension is usually acceptable because (1) the preceding the $x$ dimension access load much of the necessary data into the cache and (2) the amount of data accessed per *plane* is usually smaller than the cache. As a result, most $y$ accesses hits in the cache. However, most $z$ accesses during the depth phase suffer cache misses, which accounts for 40-70% of total execution time. Accesses along the $y$-dimension and $z$-dimension load into the cache columns of data whose length is the length of the array dimension being traversed ($y$ or $z$) and whose width is one cache line. We call each such block of data a *cache column*, one of which is highlighted in Figure 7. Each *cache column*, once loaded, is reused for as many column accesses as possible.

For FFTW, array elements are 16 bytes (a pair of double precision floating point numbers). Eight elements can fit into each 128-byte L2 cache line, so a single cache column will be able to service as many as seven adjacent column accesses before a cache miss will occur. However, if a cache column is larger than

| Input | Type | Elapsed Cycles | TLB hit rate | L1 hit rate | L2 hit rate | Miss rate | Speedup | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | Depth | Overall |
| 567x61x51 | Base | 3.1B | 98.92% | 92.66% | 4.97% | 2.37% | | |
| | Impulse | 2.2B | 99.99% | 93.50% | 5.50% | 1.00% | 2.64 | 1.40 |
| 576x57x31 | Base | 1.8B | 99.06% | 93.03% | 4.90% | 2.07% | | |
| | Impulse | 1.2B | 99.99% | 94.55% | 4.42% | 1.03% | 2.74 | 1.47 |
| 576x7x11 | Base | 36.5M | 99.46% | 92.53% | 4.70% | 2.77% | | |
| | Impulse | 18.6M | 99.98% | 93.92% | 5.48% | 0.60% | 3.38 | 2.29 |

**Table 2.** Performance results for FFTW benchmark

the L2 cache, which is the case for input arrays with large $y$ or $z$ dimensions, then almost every access during the depth phase will be a cache miss. The reason for this is that each cache line in the cache column will be evicted before it can be reused. For this benchmark, prefetching is ineffective because the amount of work performed per element is dwarfed by the time required to load a cache line from memory. Because FFTs of interest are performed on fairly large input arrays, we evaluated the performance of DCA only for such arrays where the *cache column* size exceeds cache sizes. Also, we only consider arrays with large $z$ dimensions and thus restrict our optimization to the depth phase. This makes our results conservative, as additional performance benefits could be had by applying DCA to accesses along the $y$ accesses in the first phase of the FFT. To reduce simulation overhead, we simulated an 8-kilobyte L1 cache and a 64-kilobyte 64K L2 cache. For these cache sizes, the height of a cache column to be at least 512. We arbitrarily chose input's $z$ dimension value to be 567 and 576. the $x$ and $y$ dimension sizes also were chosen arbitrarily.

The FFTW library consists of highly optimized code for computing parts of the transform, called *codelets* [6]. Every multidimensional FFT is translated into a series of calls to these codelets. As part of optimizing FFTW to exploit Impulse's DCA mechanism, we modified each codelet to create dynamic indirection vectors pointing at the array elements being accessed by that codelet. As in the random access benchmark, we unrolled-and-jammed the resulting code and added prefetching instructions to overlap computation with the DCA operation. Conventional compiler-directed prefetching or data reordering techniques will not work because the addresses and strides are input parameters for the codelets, and thus only known at runtime.

Table 2 presents our results for the FFT benchmark for various input sizes. DCA improves the performance of the depth phase by 2.64 to 3.48, depending on the input size. Total application speedup ranged from 1.40 to 2.29. The reason for these performance improvements can also be seen in Table 2. The use of DCA reduces the miss rate by more than a factor of two, and TLB performance improves significantly. The TLB performance improvement is due to the fact that all DCA accesses are to a small range of addresses, rather than the entire range of the input array.

## 6 Conclusions and Future Work

The development of the dynamic cache line assembly mechanism is still in its infancy. Future work includes applying it to more applications and further optimizing its performance. We believe the proposed mechanism can be effective for many applications with poor locality. To confirm this hypothesis, we are evaluating DCA's potential on a mix of pointer-intensive programs from the Olden benchmark suite, image processing programs, and irregular scientific application kernels (Moldyn and NBF).

The performance of dynamic cache line assembly can be improved in a number of ways. The current implementation loads each cache line of the address region from the memory before overwriting it with new addresses. If the processor we modeled had supported for *write, no-allocate*, such as is possible via the Alpha 21264 WH64 instruction [4], we could eliminate this unnecessary cache miss. Along the same lines, without support from the ISA, we must synchronize flushes of the address region and the subsequent accesses of the data region using memory barriers. A memory barrier serializes accesses before and after it and can impede the processor access streams. One way to eliminate this effect would be to extend the ISA with a special "indirection gather" instruction. The instruction would combine a flush back of the address region with a prefetch of the corresponding data region.

In conclusion, we believe that as the performance gap between processors and DRAM grows, a more flexible memory interface will be necessary to hide memory latency. Simply building larger on-chip caches will not suffice, and will increase cache access latency. We have begun investigating the potential benefits of allowing applications to selectively read/write data from/to random locations in memory efficiently when conventional caching does not suffice. Our initial experimental results have shown that for applications with poor spatial locality, such a mechanism can improve performance by a factor of three or more.

## References

1. S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, Oct. 1994.
2. J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.
3. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. Technical Report TR-542, University of Rochester, November 1994.
4. Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.
5. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the*

*1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, May 1999.

6. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP Conference*, 1998.
7. J. W. Manke and J. Wu. *Data-Intensive System Benchmark Suite Analysis and Specification.* Atlantic Aerospace Electronics Corp., June 1999.
8. MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.
9. V. Pai, P. Ranganathan, and S. Adve. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.
10. S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
11. M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, June 1998.
12. Y. Yamada. *Data Relocation and Prefetching in Programs with Large Data Sets.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.
13. L. Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000.
14. L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 98–107, Oct. 1999.

# Adaptively Mapping Code in an Intelligent Memory Architecture*

Yan Solihin[1,3], Jaejin Lee[2], and Josep Torrellas[1]

[1] University of Illinois at Urbana-Champaign
[2] Michigan State University
[3] Los Alamos National Laboratory
{solihin,torrella}@cs.uiuc.edu, jlee@cse.msu.edu
http://iacoma.cs.uiuc.edu/flexram

**Abstract.** This paper presents an algorithm to automatically map code to a generic Processor-In-Memory (PIM) system that consists of a host processor and a much simpler memory processor. To achieve high performance with this type of architecture, code needs to be partitioned and scheduled such that each section is assigned to the processor on which it runs most efficiently. In addition, processors should overlap their execution as much as possible.
Our algorithm is embedded in a compiler and run-time system and maps applications fully automatically using both static and dynamic information. Using a set of applications and a simulated architecture, we show average speedups of 1.7 over a single host with plain memory. The speedups are very close and often higher than ideal speedups on a more expensive multiprocessor system composed of two identical host processors. Our work shows that heterogeneity can be cost-effectively exploited, and represents one step toward effectively mapping code to more advanced PIM systems.

## 1 Introduction

Processor-in-Memory (PIM) chips, by integrating processor logic and memory in the same chip, enable low-latency and high-bandwidth communication between processor and memory, thereby promising high performance [2, 6, 8, 9, 12, 14, 15, 16, 17, 19, 21]. One interesting use of these chips is to replace the main memory chips in a workstation or server. In this case, PIM chips can act as co-processors in memory that execute code when signaled by the host (main) processor. This approach is taken by Active Pages [14], DIVA [6], and FlexRAM [8] among others.

This class of architectures provide a heterogeneous mix of processors: host and memory processors. Host processors are more powerful, are backed up by

* An extended version of this paper appears in [11]. This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, Michigan State University, and gifts from IBM and Intel.

deep cache hierarchies, and see a higher latency to memory. Memory processors are typically less powerful, see a lower latency to memory, and (at least in theory) are much cheaper. The question we address in this paper is: how to automatically program these architectures?

Previous work on programming these architectures [6, 4, 8, 14] manually identifies the code sections to run on memory processors. This process is not transparent to the programmer. In addition, previous work has largely focused on parallel execution of applications on only the memory processors.

In this paper, we present a compiler and run-time algorithm to automatically partition programs into sections and map each section on its most suitable processor, while maximizing the execution overlap of the host and memory processors. We apply our algorithm to both numerical and integer applications. We find that our algorithm effectively exploits the heterogeneity of the architecture.

## 2 Intelligent Memory Architecture

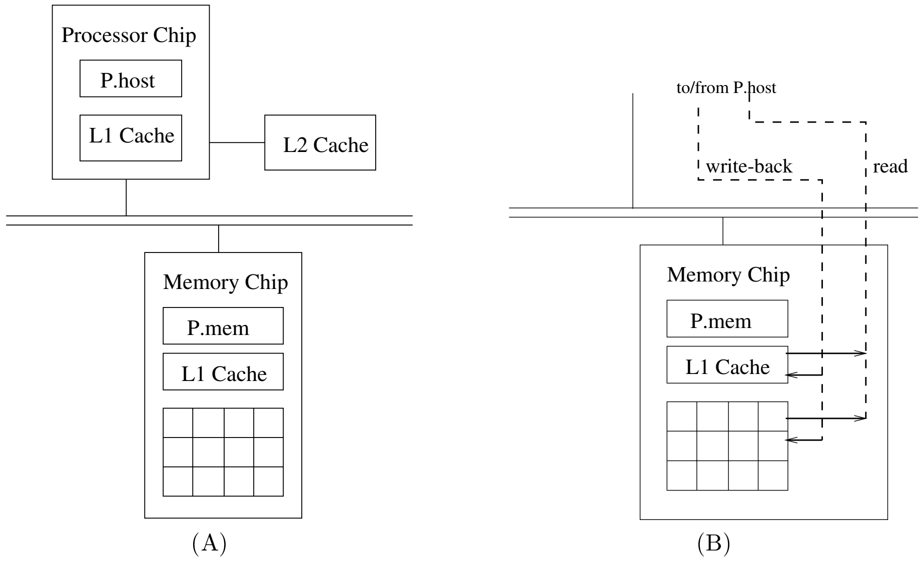


**Fig. 1.** A simple intelligent memory architecture.

To simplify the problem, in this paper we only consider the simple architecture of Figure 1(A), which has a single host processor (*P.host*) and a single memory processor (*P.mem*). We are in the process of extending our techniques to map code to architectures with multiple processors of each type.

To reduce the cost of the system, the processor and memory chips are connected with an off-the-shelf interconnection. As a result, P.mem cannot be the master of the interconnection to initiate transactions. Furthermore, there is no hardware support for cache coherence between the P.host and P.mem caches. There is, however, some simple support to make programming easier (Figure 1(B)). Specifically, when P.host writes back a line to memory, P.mem's cache is also updated if it contains a copy of the line. In addition, when P.host requests a line from memory, if P.mem's cache has a copy of it, the cache overwrites the data returning to P.host.

With this support, cache coherence can be ensured by the compiler with simple *write-back* and *invalidate* commands that control P.host's caches. The compiler inserts these commands in the program. Specifically, before transferring execution from P.host to P.mem, the P.host cache writes back any dirty lines that the P.mem may want to read. Furthermore, before returning execution to the P.host, the P.host cache invalidates any lines that the P.mem may have written.

## 3 Modules and Partitioning

Our compiler and run-time algorithm automatically maps both numeric and integer applications to the architecture of Section 2. The algorithm starts by partitioning the code into units of execution called *modules*. Modules have a homogeneous behavior in terms of computing and memory requirements, exhibit good locality, and are easy to extract from the original code. A natural and intuitive unit on which to build modules is a loop.

With *basic partitioning*, we find *basic modules*. A basic module is either a loop nest where each nesting level has only one loop, or a call to a subroutine that is composed of only one such loop nest. The loop nest may span several subroutine levels.

With *advanced partitioning*, we take the basic modules and try to expand their size or to combine several of them while keeping their behavior relatively homogeneous and enhancing locality. The result is *compound modules*. Compound modules do not have to be loops. They are generated by merging basic modules with nearby statements that access overlapping data sets. In addition, they are also generated by combining adjacent basic modules that we expect to have the same *affinity*. We say that a module has affinity for P.host or P.mem if it runs faster on P.host or P.mem, respectively.

To estimate the affinity of a basic module we use two approaches: for integer applications, we use a profiling run with a different input set, while for numeric applications, we use Delphi's static performance predictor [3]. The latter estimates the compute and cache miss times of the module in each processor.

## 4 Adaptive & Overlapped Execution

The basic or compound modules identified in Section 3 must now be scheduled for execution on either P.host or P.mem. The schedule can be decided statically, based on the affinity estimated by either the static predictor or the profile. This approach we call Static.

A more advanced approach is to decide the schedule adaptively at run time. In this case, the compiler inserts instrumentation code that measures, at run time, the execution time of some invocations of the module (or, if applicable, some of its iterations) on P.host and some on P.mem. Based on these measurements, the run-time system schedules subsequent module invocations (or iterations). The different dynamic scheduling strategies supported are shown in Table 1.

**Table 1.** Different dynamic scheduling strategies.

| Name | What We Do | Note |
|---|---|---|
| *Coarse Basic* (Coarse) | First invocation of module runs on P.host. Second one runs on P.mem. The processor that ran the fastest is assigned the module for the rest of the invocations in the application. | |
| *Coarse Most Recent* (CoarseR) | First invocation runs on P.host. Second one runs on P.mem. From now on, after every invocation, we compare the execution time to the most recent execution time on the *other* processor. Based on the result, the subsequent invocation is scheduled on the processor that ran the fastest. | It can adapt to changes in the behavior of the module across invocations. |
| *Fine Basic* (Fine) | For each invocation of the module, repeat the following. First iteration runs on P.host. Second one runs on P.mem. The processor that ran the fastest is assigned the rest of the iterations in the invocation. | It only works for modules that have an all-enclosing loop. It may have high overhead. |
| *Fine First Invocation* (FineF) | First iteration of first invocation runs on P.host. Second iteration of first invocation runs on P.mem. The processor that ran the fastest is assigned the rest of the iterations in this invocation and the rest of the invocations. | It only works for modules that have an all-enclosing loop. It has the lowest overhead, but it may produce a wrong prediction. |

The different strategies in the table may be best under different situations. Table 2 lists the best strategy based on how the behavior of the module execution varies across invocations of the module and across iterations of a given invocation of the module.

Finally, to further speed-up code execution, we overlap the computation of P.host and P.mem. To this end, we divide the application into two classes of regions: *module-wise parallel regions*, where there are multiple modules that can be run in parallel with respect to one another, and *module-wise serial regions*, where only one module can be run at a time because of dependences between modules. Note that here we use basic partitioning because it creates simpler modules and, therefore, exposes more parallelism.

**Table 2.** Comparing the different dynamic scheduling strategies.

| Num. Invocations | Behavior Across Invocations | Behavior Across Iterations | Best Strategy |
|---|---|---|---|
| $> 2$ | Constant | Constant | Fine first invocation |
| | Constant | Variable | Coarse basic |
| | Variable | Constant | Coarse most recent, Fine basic |
| | Variable | Variable | Coarse most recent |
| 1 or 2 | Constant | Constant | Fine first invocation |
| | Constant | Variable | — |
| | Variable | Constant | Fine basic |
| | Variable | Variable | — |

In the module-wise parallel regions, we simply overlap the execution of the modules on the different processors. In a module-wise serial region, we try to split the module into two pieces, one for P.host and one for P.mem. Specifically, if the module is a fully-parallel loop, we divide the iteration count into two unequal chunks that are expected to take equal time to execute. Otherwise, we try to use loop distribution across the two processors with or without synchronization. If none of these techniques is possible, we simply apply the best sequential scheduling strategy as described above.

In all cases of overlapped execution, to decide how to partition the work between P.host and P.mem, we can use either static-only information or dynamic information. We call these cases OverSta and OverDyn, respectively.

## 5 Evaluation Setup

The code generated by the compiler is targeted to a MINT-based [20] simulation environment [10]. The simulation environment can model dynamic superscalar processors with register renaming, branch prediction, and non-blocking memory operations [10]. The architecture modeled is that of Section 2, with a bus connecting the processor and memory chips. The architecture is modeled cycle by cycle, including contention effects. Table 3 shows the parameters used for each component of the architecture. The L2 cache size used is 1 Mbyte, except for one of the applications (Bzip2), which is simulated with a 512-Kbyte L2.

Our choice of P.mem's clock frequency is motivated by recent advances in Merged Logic DRAM process. They seem to enable the integration of on-chip logic that cycles as fast as in a logic-only chip, with DRAM memory that is only 10% less dense than in a DRAM-only chip [13, 7].

The table also includes the overheads involved in invalidating and writing back lines from P.host's L2 cache. We assume the following hardware support in the L2 cache controller. Suppose that we want to write back *num_cache_lines* lines. To program the controller, P.host suffers an overhead of $5{+}1{\times}$*num_cache_li* cycles. Then, the controller writes back the desired lines in the background without stalling P.host. Note, however, that the write backs must be completed before passing execution to P.mem.

**Table 3.** Parameters of the simulated architecture. Cache and memory latencies correspond to contention-free round-trips from the processor.

| Module | Parameter | Value |
|---|---|---|
| P.host | Frequency | 800 MHz |
| | Issue Width | Out-of-order 6-issue |
| | Func. Units | 4 Int + 4 FP + 2 Ld/St units |
| | Pending Ld/St | 8/16 |
| | Branch Penalty | 4 cycles |
| P.mem | Frequency | 800 MHz |
| | Issue Width | In-order 2-issue |
| | Func. Units | 2 Int + 2 FP + 1 Ld/St units |
| | Pending Ld/St | 4/4 |
| | Branch Penalty | 2 cycles |
| P.host Caches | L1-Data | Write-through, 32-KB, 2-way, 32-B line, 2-cycle hit |
| | L2-Data | Write-back, 1-MB (512-KB for Bzip2), 4-way, 128-B line, 10-cycle hit |
| | Write-Back Overhead | $5 + 1 \times num\_cache\_lines$ cycles to program. Actual write back of data occurs in background |
| | Invalidation Overhead | $5 + 1 \times num\_cache\_lines$ cycles total. It is typically overlapped with P.mem's execution |
| P.mem Cache | L1-Data | Write-back, 16-KB, 2-way, 32-B line, 2-cycle hit |
| Memory & Bus | Memory Latency | If row buffer miss: 160 cycles from P.host & 21 cycles from P.mem<br>If row buffer hit: 152 cycles from P.host & 13 cycles from P.mem |
| | Bus Type | Split transaction, 16-B wide |
| | DRAM Memory Size | 64 MB per chip |

Assume now that we want to invalidate *num_cache_lines* lines. In this case, P.host suffers a total overhead of $5+1\times num\_cache\_lines$ cycles. Note that these cycles can be overlapped with P.mem execution. This overhead only affects the final execution time if the invalidations have not completed when P.mem finishes execution and P.host is expected to resume.

P.host and P.mem synchronize at module boundaries. Specifically, P.host writes to a register in P.mem to signal P.mem that it can begin execution. When P.mem has completed execution, it writes to another one of its registers so that P.host can see it. The overheads involved in these synchronizations are considered in our simulation.

The applications evaluated are numerical and integer programs: Swim and Mgrid from SPECfp2000 [5], Tomcatv from SPECfp95 [5], LU from [18], TFFT2 from NAS [1], and Bzip2 from SPECint2000 [5]. All floating-point applications use double precision. Table 4 shows the data set sizes used and what the applications are computing. For Bzip2, we perform the runs with three different input data sets: the Input1 runs use the Train input set, while the Input2 runs use this paper in postscript format as input. The two runs of Bzip2 use the profile generated with the third (*Test*) input set.

**Table 4.** Applications used.

| Application | Data Size and Number of Iterations | Description |
|---|---|---|
| Swim | 513 × 513, 10 iterations | Shallow water simulation |
| Tomcatv | 513 × 513, 5 iterations | Vectorized mesh generation |
| LU | 512 × 512 | LU matrix decomposition |
| TFFT2 | $2^{17}$ elements, 5 iterations | Fast fourier transformation |
| Mgrid | 64 × 64 × 64 grid, 3 iterations | Multi-grid solver: 3D potential field |
| Bzip2 | Input1: Train<br>Input2: This paper in postscript | Compression and decompression algorithm (level 7) |

## 6 Evaluation Results

Table 5 shows the characteristics of the basic modules obtained by our compiler. They account for 97% of the execution time on average. The table also shows that different applications have a different distribution of affinity for P.host and P.mem.

**Table 5.** Characteristics of the basic modules that are called at least once.

| Characteristic (% of P.host Time) | Swim | Tomcatv | LU |
|---|---|---|---|
| Total Modules | 16 (100.00%) | 7 (96.46%) | 5 (99.99%) |
| Parallel Modules | 16 (100.00%) | 5 (56.16%) | 3 (7.36%) |
| Serial Modules | - | 2 (40.30%) | 2 (92.63%) |
| Modules with P.host Affinity | 1 (6.54%) | 3 (18.16%) | 2 (92.63%) |
| Modules with P.mem Affinity | 15 (93.46%) | 4 (78.30%) | 3 (7.36%) |
| Average Invocations per Module | 5.7 | 5.0 | 409.4 |
| **Characteristic (% of P.host Time)** | **TFFT2** | **Mgrid** | **Bzip2 (*Test*)** |
| Total Modules | 17 (99.28%) | 21 (99.79%) | 75 (85.37%) |
| Parallel Modules | 15 (93.93%) | 18 (96.85%) | 24 (0.68%) |
| Serial Modules | 2 (5.35%) | 3 (2.94%) | 51 (84.69%) |
| Modules with P.host Affinity | 8 (44.04%) | 6 (23.91%) | 51 (56.41%) |
| Modules with P.mem Affinity | 9 (55.24%) | 15 (75.88%) | 24 (28.96%) |
| Average Invocations per Module | 1688.5 | 105.7 | 73,499.0 |

Figure 2 and Figure 3 show the execution time for each application. In each chart, the two leftmost bars correspond to running the application on P.host alone (P.host(alone)) and on P.mem alone (P.mem(alone)). Then, there are nine pairs of two bars, where each pair corresponds to a different way of partitioning and scheduling. A given pair shows the execution time of P.host and P.mem (which are necessarily the same) broken down into several categories. The pairs of bars correspond to static scheduling (Static), dynamic sequential scheduling with basic modules (Coarse, CoarseR, Fine, FineF), dynamic sequential scheduling with compound modules (AdvCoarse, AdvCoarseR), and overlapped scheduling (OverSta, OverDyn). For Bzip2, since we have two input sets, we only show a subset of the bars.

**Non-Overlapped Execution**

P.host(alone) and P.mem(alone) show that the relative emphasis on computing and memory activity varies across applications: Swim, Tomcatv, and Mgrid run faster on P.mem, while LU runs faster on P.host and TFFT2 runs equally fast on both processors. Depending on the input set, Bzip2 can be much faster on P.host or almost as fast in P.host as in P.mem. Overall, these bars show that neither P.host nor P.mem is the best place to run all and every application. If an application executes on the less optimal processor, it may take up to 100% longer to run.

Static schedules modules in the floating-point applications according to the static predictor; if the latter cannot estimate the affinity, the module runs on P.host. From the figure, we see that Static performs relatively well. It runs quite fast for Swim, LU, and TFFT2. Overall, Static is attractive because of its simplicity. However, the static predictor does not currently analyze the complicated code in the integer application (Bzip2). Consequently, Static in Bzip2 uses profiling information, which does not necessarily perform well. Specifically, when the input set is very different from the input set used for profiling, Static performs poorly, as it is shown with Input2 in Bzip2.

Coarse and CoarseR tend to be good choices. They are usually as fast or faster than Static. The reason is that they adaptively run the modules on the processors for which the modules have the true affinity. In the process of doing so, however, they are likely to run each module sub-optimally at least once.

Coarse and CoarseR behave similarly for Swim, Tomcatv, Mgrid, and Bzip2 (Input2). In these applications, the workload in a given module tends to remain constant across invocations. As a result, CoarseR does not offer any advantage over Coarse. However, in LU, TFFT2, and Bzip2 (Input1), the workload in a given module varies across invocations. The variation of the modules in LU and Bzip2 (Input1) is gradual, which means that CoarseR can adapt well. The result is that CoarseR is about 20% faster than Coarse in LU. In TFFT2, however, the workload of the largest module varies abruptly, with peaks at every 8 invocations. After one of these abrupt peaks is recorded on a processor, the module is scheduled on the other processor for many subsequent invocations, even though it would run faster on the first processor. This behavior disrupts the smooth execution of the CoarseR algorithm, slowing it down slightly, relative to Coarse.

The fine strategies are not as attractive. Specifically, Fine is sometimes slow because of the high overhead resulting from its frequent decision runs (TFFT2). As for FineF, although it has the lowest decision run overhead of all the dynamic schemes, it often suffers because all decisions are made based exclusively on the

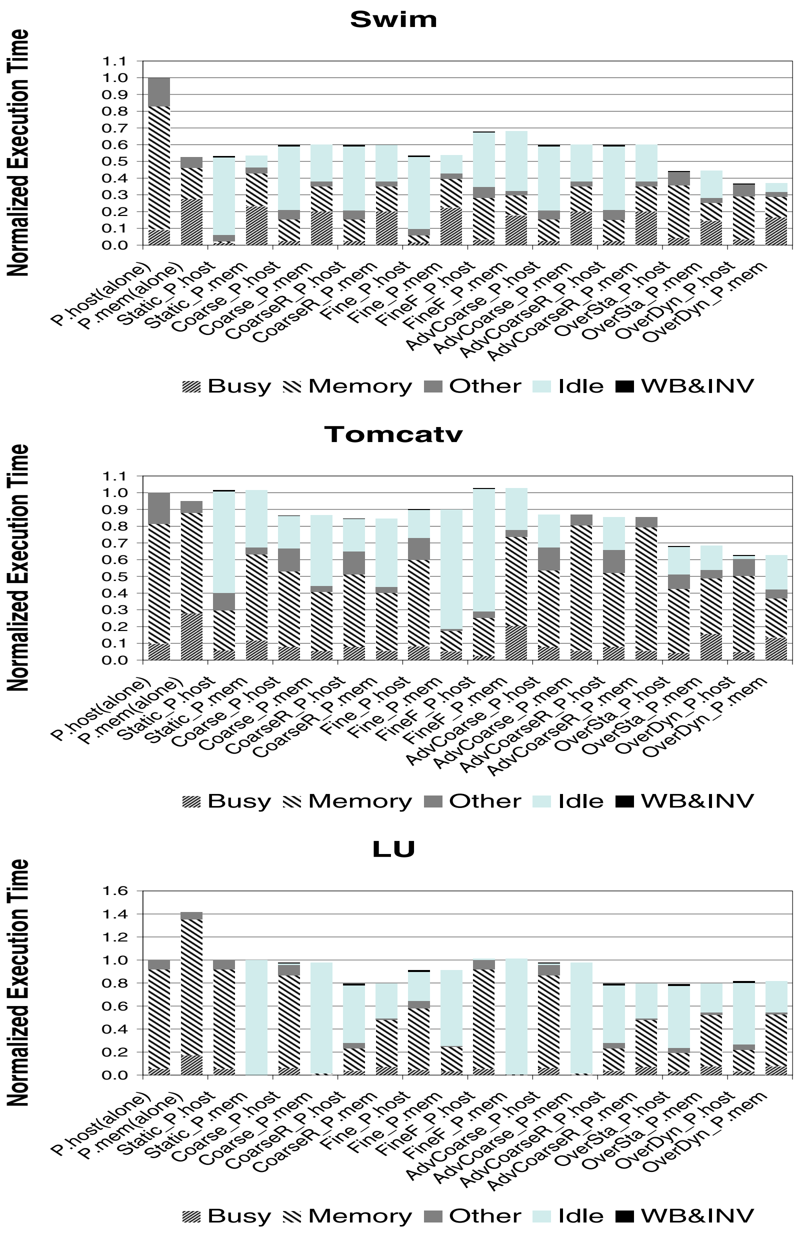


**Fig. 2.** Execution time of the applications. The bars are divided into execution of instructions (Busy), stall due to memory accesses (Memory), stall due to pipeline hazards (Other), time waiting for the other processor (Idle), and time spent writing back or invalidating lines in the caches of P.host (WB&INV).

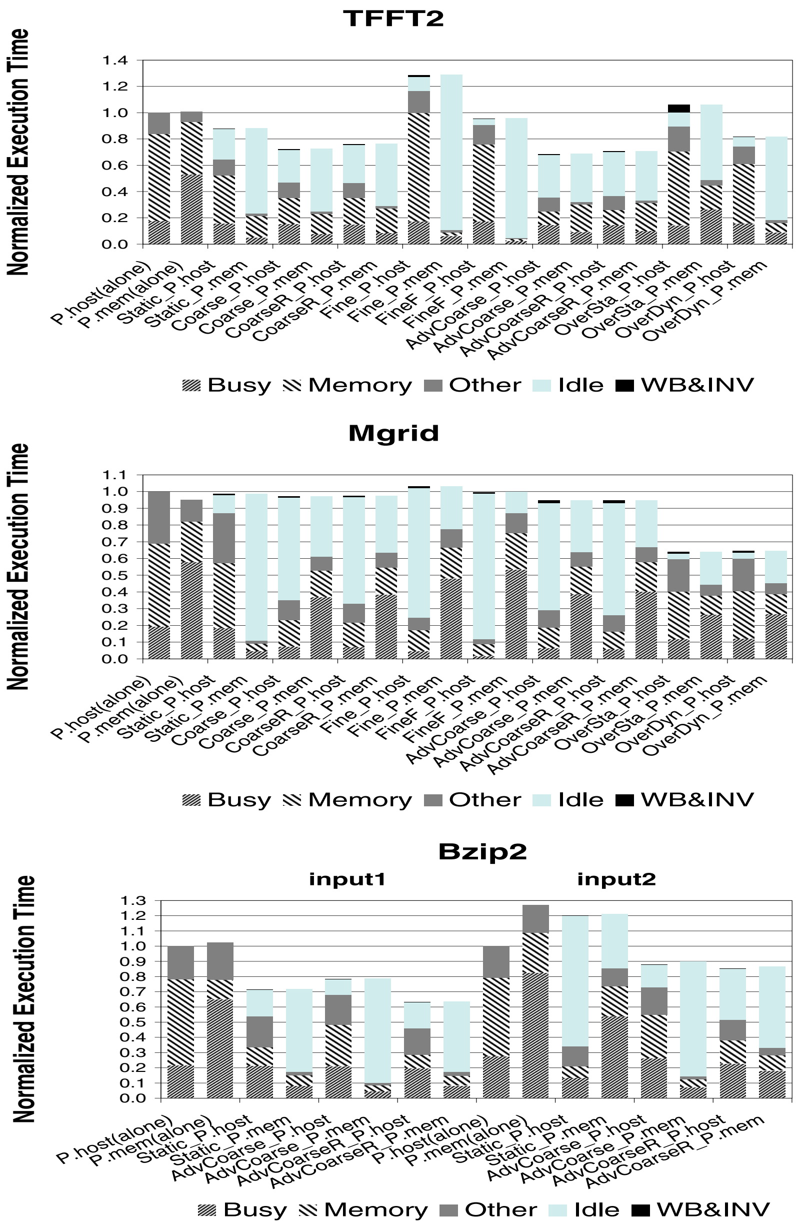


**Fig. 3.** Execution time of the applications. The bars are divided into execution of instructions (Busy), stall due to memory accesses (Memory), stall due to pipeline hazards (Other), time waiting for the other processor (Idle), and time spent writing back or invalidating lines in the caches of P.host (WB&INV).

first two iterations of the first invocation of the module. Consequently, unless the workload of the module is constant across invocations and iterations, the decision is likely to be sub-optimal (Tomcatv).

The figures show that advanced partitioning has little impact on the performance of numeric applications. Indeed, the AdvCoarse and AdvCoarseR bars are similar to the Coarse and CoarseR ones. The reason is that basic modules are already large enough to dwarf scheduling and other overheads. However, advanced partitioning is effective for Bzip2. Although not shown in the figure, AdvCoarse and AdvCoarseR are significantly faster than Coarse and CoarseR for Bzip2. The reason is that basic modules are quite small in Bzip2 and, therefore, overheads are relatively large.

Overall, we conclude that, among the non-overlapped execution schemes, AdvCoarseR is the best. It is on average 23% faster than P.host(alone) and 20% faster than P.mem(alone).

**Overlapped Execution**

Overlapped execution speeds up the application significantly in 3 out of 6 applications. Specifically, in Swim, Tomcatv, and Mgrid, the overlapped schemes OverSta and OverDyn speed up the application by 30-40% relative to AdvCoarseR. With these schemes, we are utilizing processor resources that would otherwise remain idle. In LU and Bzip2 (not shown), the overlapped schemes have no impact over AdvCoarseR. The reason is that the most significant modules in these codes have dependences that prevent them from being partitioned following the algorithm of Section 4.

In TFFT2, however, overlapped execution is noticeably slower than AdvCoarseR. The reason is that overlapped scheduling induces extra overheads on P.host. Specifically, it causes extra instruction execution and more cache misses on P.host, all to ensure data coherence. The extra instructions are necessary to write back and invalidate cached data structures when execution is transferred to P.mem and back (WB&INV in Figure 3) and to generate the addresses of these data structures (higher Busy in Figure 3). The extra misses occur when, after the cache invalidations, the data is reloaded into P.host's cache (higher Memory Stall in Figure 3).

Interestingly, the chart for TFFT2 shows that, while OverSta suffers greatly from these overheads, OverDyn is able to eliminate most of them and only takes 12% longer than AdvCoarseR. OverDyn is better because it is aware of the extra overheads involved with overlapped execution and schedules modules more conservatively.

Overall, taking the average over all applications, OverDyn is 18% faster than AdvCoarseR, the best non-overlapped scheme. Consequently, OverDyn is our best scheme of all and, therefore, overlapped execution is our choice.

**Summary**

As a summary, Table 6 compares the speedups obtained by AdvCoarseR and OverDyn, and the *ideal* Amdahl's speedups for a machine that has 2 P.host processors with plain memory. From the table, we can see that, by running each section of the code on the processor where we expect it to run best, OverDyn delivers an average speedup of 1.7 over a single host with plain memory. The application speedups are very close and often higher than the *ideal* speedups on a more expensive multiprocessor system composed of two identical host processors.

**Table 6.** Comparing speedups. In averaging the speedups, each input of Bzip2 is given half weight.

| Application | $\frac{P.host(alone)}{AdvCoarseR}$ | $\frac{P.host(alone)}{OverDyn}$ | Ideal Using 2 P.hosts |
|---|---|---|---|
| Swim | 1.67 | 2.71 | 2.00 |
| Tomcatv | 1.17 | 1.60 | 1.67 |
| LU | 1.26 | 1.22 | 1.04 |
| TFFT2 | 1.42 | 1.22 | 1.91 |
| Mgrid | 1.05 | 1.55 | 1.94 |
| Bzip2 (Input1) | 1.57 | - | 1.00 |
| Bzip2 (Input2) | 1.17 | - | 1.00 |
| Average | 1.32 | 1.66 | 1.59 |

## 7 Conclusions

We presented and evaluated a compiler and run-time algorithm to automatically partition and map code to a simple intelligent memory architecture. A major conclusion is that some applications run best on the sophisticated P.host, while others run best on the simpler P.mem. Furthermore, the same is true for different code sections within an application. Overall, our results indicate that a heterogeneous mix of processors is a promising approach to speed-up applications cost-effectively. The cost-effectiveness of such architectures is likely to be higher than that of a conventional multiprocessor system with homogeneous processors. We are in the process of extending our algorithms to architectures with several P.hosts and several P.mems.

## 8 Acknowledgments

We thank D. Padua, A. Hoisie, C. Cascaval, P. Wu, M. Cintra, and anonymous reviewers for useful discussion and feedback on the draft version of this paper. We also thank M. Huang, J. Renau, and C. Cascaval for help with the software used in this study.

## References

[1] NAS Parallel Benchmark. http://www.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-98-009/.

[2] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[3] C. Cascaval, L. DeRose, D. A. Padua, and D. Reed. Compile-Time Based Performance Prediction. In *Twelfth International Workshop on Languages and Compilers for Parallel Computing*, 1999.

[4] J. Chame, J. Shin, and M. Hall. Compiler Transformations for Exploiting Bandwidth in PIM-Based Systems. In *Solving the Memory Wall Problem Workshop*, June 2000.

[5] Standard Performance Evaluation Corporation. http://www.spec.org.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In *Supercomputing 1999 (SC99)*, November 1999.

[7] S. S. Iyer and H. L. Kalter. Embedded DRAM Technology: Opportunities and Challenges. *IEEE Spectrum*, April 1999.

[8] Y. Kang, M. Huang, S. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the International Conference on Computer Design*, October 1999.

[9] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[10] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[11] J. Lee, Y. Solihin, and J. Torrellas. Automatically Mapping Code in an Intelligent Memory Architecture. In *International Symposium on High Performance Computer Architecture*, January 2001.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture*, June 2000.

[13] IBM Microelectronics. Blue Logic SA-27E ASIC. In *News and Ideas of IBM Microelectronics*, February 1999. http://www.chips.ibm.com/news/1999/sa27e.

[14] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture*, pages 192–203, June 1998.

[15] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. Farrens, and A. Chopra. Exploiting ILP in Page-Based Intelligent Memory. In *International Symposium on Microarchitecture*, November 1999.

[16] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick. A Case for Intelligent DRAM. In *IEEE Micro*, pages 33–44, March/April 1997.

[17] D. Patterson and M. Smith. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. 1997.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in Fortran 77. Cambridge University Press, 1992.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *International Symposium on Microarchitecture*, November 1998.

[20] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *MASCOTS'94*, pages 201–207, January 1994.

[21] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.

# The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems*

Richard C. Murphy, Peter M. Kogge, and Arun Rodrigues

Department of Computer Science and Engineering
University of Notre Dame
{rcm,kogge,arodrig6}@cse.nd.edu

**Abstract.** Processing-In-Memory (PIM) circumvents the von Neumann bottleneck by combining logic and memory (typically DRAM) on a single die. This work examines the memory system parameters for constructing PIM based parallel computers which are capable of meeting the memory access demands of complex programs that exhibit low reuse and non uniform stride accesses. The analysis uses the Data Intensive Systems (DIS) benchmark suite to examine these demanding memory access patterns. The characteristics of such applications are discussed in detail. Simulations demonstrate that PIMs are capable of supporting enough data to be multicomputer nodes. Additionally, the results show that even data intensive code exhibits a large amount of internal spatial locality. A *mobile thread* execution model is presented that takes advantage of the tremendous amount of internal bandwidth available on a given PIM node and the locality exhibited by the application.

## 1 Introduction and Motivation

Processing-in-Memory (PIM)[14,13,4] (also known as Intelligent RAM [21], embedded RAM, or merged logic and memory) systems exploit the tremendous amounts of memory bandwidth available for intra-chip communication, and therefore circumvent the von Neumann bottleneck, by placing logic and memory (typically DRAM) on the same die. This technology allows for the construction of highly distributed systems, but with a very large latency gap between high speed local memory macro accesses and remote accesses. The construction of high performance systems incorporating PIMs must successfully exploit

* An early part of this effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under Cooperative Agreement number F30602-98-2-0180 as part of the Data Intensive Architecture Project under the Data Intensive Program. The U.S. Government is authorized to reproduce and distribute for Governmental purposes notwithstanding any copyright annotation thereon.
The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

the bandwidth available for on-chip accesses while simultaneously tolerating very long remote access latencies. Multi-threading, similar to that used in the Tera[1], seems the natural method for tolerating remote accesses, however, such a model does not inherently take advantage of the relatively large amount of quickly accessed memory available on a PIM node. In fact, the Tera generally requires about the same amount of persistent state available in the L1 cache of a modern microprocessor[1], and a typical PIM node is likely to have 3 to 4 orders of magnitude more memory available.

This paper describes the memory access behavior of several canonical *data intensive* applications (that is, applications which exhibit frequent data accesses in a highly irregular pattern, and low reuse). These applications, which are particularly difficult for most modern architectures to accommodate, represent scientific problems of significant interest. Thus, the ability to successfully cope with their requirement will yield tremendous insight beyond the more simplistic benchmarks used today.

The characterization of these memory workloads is determined using a **single threaded** trace generated from actual program execution. This represents the first step in modeling a multi-threaded system and identifying a simple data-placement scheme.

This paper is organized as follows: Section 2 describes the benchmarks and the rational for for choosing the Data Intensive Systems suite. Section 3 provides an overview of PIM technology and the general assumptions behind the system simulated. Section 4 enumerates the simulation methodology and describes the desirable outcome of simulation (that is, the condition of success). Section 5 describes the mechanism for analysis, particularly focusing on the Cumulative Instruction Probability Density (CIPD), which will indicate the measure of the degree of success. Section 6 provides the results of experimentation which determine both the size and form of a well constructed working set. Section 7 describes the simulation of a *mobile thread* model of computation in which a thread travels throughout the system looking for the data it needs, as well as the costs and benefits of such a model. Finally, Section 8 contains the conclusions and a description of future work.

Further details on the experimentation described in this paper, as well as a complete set of results for all the benchmarks can be found in [18].

## 2 Benchmarks

This work concentrates on the analysis of the Data Intensive Systems (DIS) benchmark suite[2,3]. These benchmarks are atypical in that their memory access patterns exhibit a low degree of reuse and non-linear stride. Thus the focus will naturally be on the performance of the memory system over that of the processing elements. Clearly in the case of PIM the interaction between the demand for data and its supply is the preeminent characteristic under study. Most benchmark suites, in sharp contrast, are designed to be quickly captured in a processor's cache so as to measure raw computation power. This is somewhat misleading since the performance of most modern architectures is determined by that of the memory system.

Early work focused on the performance of the SPEC95[20] integer and floating point benchmarks. The results of those experiments tended to be unenlightening as the memory access patterns were both regular and easily accommodated by even a small PIM (which has significantly more persistent state than modern caches). Tests in which the data set sizes were increased did not fare much better in that the benchmarks themselves tend to use data with a high degree of both spatial and temporal locality.

Significant research was then undertaking using the *oo7* database benchmark[6] with the underlying implementation by Pedro Diniz at USC's Information Sciences Institute, which proved significantly more interesting in that it uses more irregular data structures. Finally, with the release of the DIS suite, which includes a data management benchmark, a sufficient number of distinct data intensive applications were available as a coherent benchmark to allow for meaningful comparison amongst complex applications.

Additional experimentation was performed using a simple Molecular Dynamics simulation[12], which is of significant interest given its highly complex memory access patterns and IBM's Blue Gene project which will use PIM technology for similar protein folding applications. For reasons of brevity, that experimentation will not be summarized here, but can be found in [18].

The DIS suite is composed of the following benchmarks:

– **Data Management:** implements a simplified object-oriented database with an R-Tree indexing scheme [11,16]. Three operations are supported: *insert*, *delete*, and *query*. For the purposes of these experiments, only the *query* operation was examined.
– **FFT:** is a Three Dimensional Fourier Transform which uses the FFTW library for optimization. This operation could have been included as the first step in both the Ray Tracing and Method of Moments benchmarks, however given the code's relatively common use, it is treated separately. (Both the Ray Trancing and Method of Moments benchmarks take data already converted into Fourier space.)
– **Method of Moments:** represents algorithms which are frequency domain techniques for computing electro-magnetic scattering from complex objects. Typical implementations employ direct linear solves, which are highly computation intensive and can only be applied to reasonably low frequency problems. The faster solvers applied in this benchmark are memory bound since reuse is extremely low and access patterns exhibit non-uniform stride. This benchmark is derived from the Boeing implementation of fast iterative solvers for the Helmholtz equation [8,10,9].
– **Image Understanding:** attempts to detect and classify objects within a given image. This implementation requires three phases: morphological filtering, in which a spatial filter is created and applied to remove background clutter; determination of the region of interest; and feature extraction.

– **Ray Tracing:** is a component of Simulated SAR benchmark, and represents the computational core. This portion of the program consists of sending rays from a fixed point and determining where they interact with other objects.

## 3 PIM Technology and Architecture

Modern processors require that tremendous amounts of data be provided by the system's memory hierarchy, which, is becoming increasingly difficult to supply. The core of this problem, known as the *von Neumann Bottleneck* relates to the separate development of processing and memory technologies, and the different emphasis placed on each. Processors, built around logic fabrication processes which emphasize fast switching, generally follow Moore's law, while memories emphasize high density but relatively low data retrieval rates. The interconnection mechanism between the two is a narrow bus which cannot be greatly expanded due to the physical limit on the number of available pins and high capacitance of inter-chip communication.

Developments in VLSI technology, such as the trench capacitor compatible with a logic process developed at IBM, now allow for fabrication facilities which offer both high performance logic and high density DRAM on the same die. These PIMs further allow for the creation of much higher bandwidth interconnection between local memory macros and logic since it all occurs on chip.

Several proposals exist which attempt to fully utilize the potential of these fabrication developments. The IRAM project [21] at Berkeley seeks to place a general purpose core with vector capabilities along with DRAM onto a die for embedded applications. Cellular phones, PDAs, and other devices requiring processing power and relatively small amounts of memory could benefit tremendously from this type of system, even if one only considers the potential advantages in power consumption. Others, such as members of the Galileo group[5] at the University of Wisconsin see PIM as having tremendous potential in standard workstations where the on chip memory macros would become all or part of the memory hierarchy. More recently, the Stanford Smart Memories project[17] began exploring the construction of single chip systems capable of supporting a diverse set of system models.

The DIVA project [13] is currently investigating system and chip level implementations for PIM arrays functioning as part of the memory hierarchy in a standard workstation. Finally, the HTMT[22,15] project is a multi-institutional effort to construct a machine capable of reaching a petaflop or above in which a large part of the memory hierarchy consists of PIMs being designed by the Notre Dame PIM group. This portion of the memory hierarchy is a huge, two-level, multi-threaded array.

Figure 1 show a typical single node PIM layout. In the case of the target ASAP Architecture[19], a vector processor (capable of operating on 256 bit vectors in 8, 16, or 32 bit chunks) is tightly coupled with a set of memory macros. For the purposes of simulation, it is assumed that the memory macro provides 2 k-bits of data per operation through a single open row register. The ASAP's

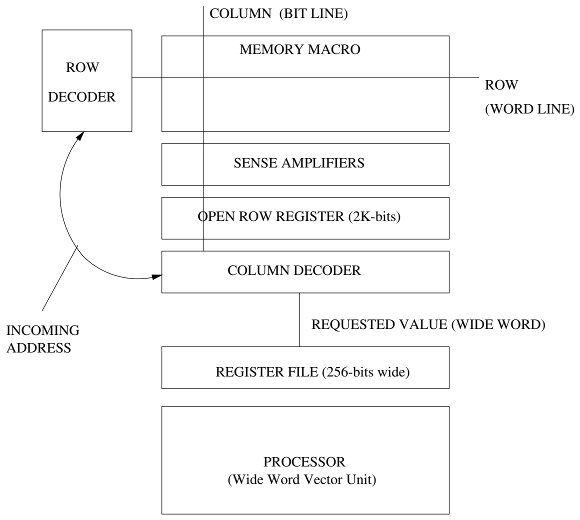

COLUMN (BIT LINE)
ROW DECODER
MEMORY MACRO
ROW (WORD LINE)
SENSE AMPLIFIERS
OPEN ROW REGISTER (2K-bits)
COLUMN DECODER
INCOMING ADDRESS
REQUESTED VALUE (WIDE WORD)
REGISTER FILE (256-bits wide)
PROCESSOR (Wide Word Vector Unit)

**Fig. 1.** Typical PIM Memory Layout

register file then accesses that data in 256 bit chunks. Thus, while a random read from memory will cause a DRAM access, a read contained in the current open row does not incur that penalty (because it is simply a register transfer operation).

The array of PIMs simulated is assumed to be homogeneous. Furthermore, for the purposes of this paper, no particular interconnection topology is assume (rather, communication events are merely counted). Experimentation over various topologies can be found in [18]. In actuality, a PIM array is likely to be heterogeneous (potentially consisting of PIMs of different types – SRAM and DRAM – and different sizes), and the interconnection network hierarchical. Multiple nodes may be present on a chip, facilitating significantly faster on-chip communications mechanisms. Additionally, since PIM systems may be part of a larger memory hierarchy, additional non-PIM processing resources or memory may be available.

PIMs, in our model, communicate through the use of *parcels*, which are messages possessing intrinsic meaning directed at named objects. Rather than merely serving as a repository for data, parcels carry distinct high level commands and some of the arguments necessary to fulfill those commands. Low level parcels (which may be handled entirely by hardware) may contain simple memory requests such as: "access the value X and return it to node K." Higher level parcels are more complicated and may take the form "resume execution of procedure Y with the following partially computed result and return the answer to node L." Thus, it should be assumed that parcels can perform both communication and computation, and may be invoked by the user, run-time system, or hardware.

## 4 Simulation Methodology

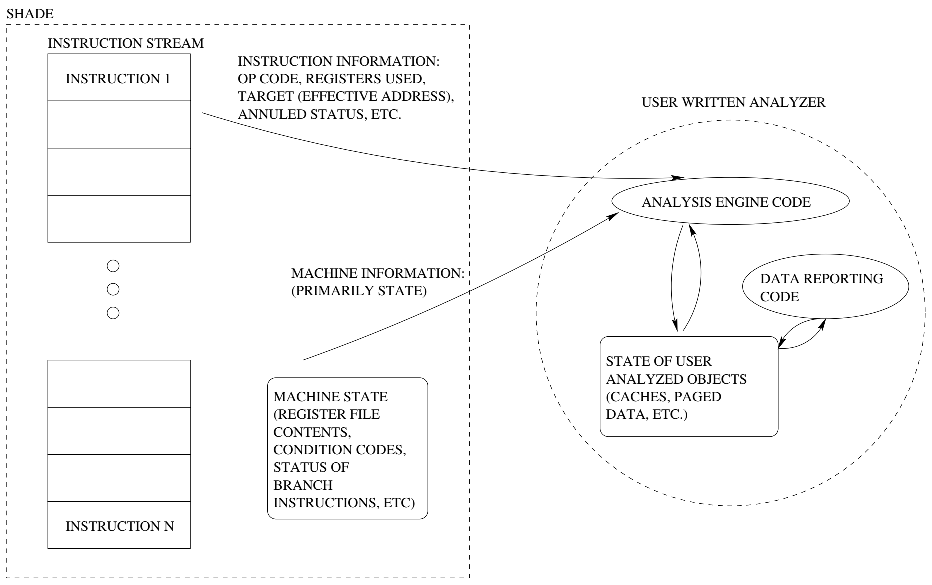

SHADE
INSTRUCTION STREAM
INSTRUCTION 1
INSTRUCTION N
INSTRUCTION INFORMATION: OP CODE, REGISTERS USED, TARGET (EFFECTIVE ADDRESS), ANNULED STATUS, ETC.
MACHINE INFORMATION: (PRIMARILY STATE)
MACHINE STATE (REGISTER FILE CONTENTS, CONDITION CODES, STATUS OF BRANCH INSTRUCTIONS, ETC)
USER WRITTEN ANALYZER
ANALYSIS ENGINE CODE
DATA REPORTING CODE
STATE OF USER ANALYZED OBJECTS (CACHES, PAGED DATA, ETC.)

**Fig. 2.** Shade Simulations

The principle benchmarking mechanism presented in this paper is the Shade suite[24] developed by Sun Microsystems. This tool allows for the analysis of any SPARC binary by providing a simple mechanism for examining the code's execution instruction by instruction. Figure 2 shows the simulation mechanism. User written analysis code takes the running instruction stream and current machine state to track the state of the processing and memory systems for a PIM array. Of particular interest are memory events, such as opening a new row or generating an off chip memory access.

Since the Shade suite traces SPARC instructions, the simulated ISA corresponds roughly to that of a typical RISC machine. This obviously does not represent the vector ASAP ISA, however, this work is primarily concerned with the performance of the memory system.

Shade does not provide a mechanism for tracing multi-threaded code, though a package to do so is under development and will be incorporated into future work. Consequently the instruction streams analyzed here are **single threaded**. However, since they are taken from the program's main loop of execution, they are not atypical.

To allow the simulation to be tractable, input sets were restricted to the 100-500 MB range, as appropriate for the particular benchmark. Additionally, simulation was limited to a 32-bit address space. Data sets were divided into three parts: code (as indicated by portions of memory subject to an instruction fetch), the stack (which grows down from the top of the address space), and the heap (everything else). For the purposes of data movement, only objects in the heap were analyzed.

Many of the simulations, though consisting of smaller data sets, were performed with an eye towards very large machines (consisting potentially of a million or more nodes). Thus, large parcel sizes (for pages, code, state, etc.), which can be handled by the extremely high bandwidth interconnection networks of such a machine, are not considered detrimental to performance. On the other hand, broadcasts, updating many remote data structures, or overhead data structures which envelop most of the memory on a given node are considered detrimental to performance.

Of particular interest is the amount of time a given thread of execution can continue on a node before an off node memory access is generated. Thus, the execution model favors uninterrupted execution for long periods of time.

## 5 Metrics

There are primarily two metrics which will be presented throughout the rest of this paper. The first, and simplest to understand, is the *miss rate*. It is, quite simply, the fraction of accesses which cause a miss over the number of accesses during the entire program execution. If $A$ represents the total number of accesses and $M$ represents the total number of misses, the miss rate is merely $\frac{M}{A}$. This is the traditional metric presented when examining the "efficiency" of caches.

However, since the measure of efficiency for the purposes of these experiments is run length between misses (off node accesses), the more detailed *Cumulative Instruction Probability Density*, or CIPD, is also presented. The CIPD is computed by dividing a program's execution up into streams of instructions for which no miss is generated, given the memory state of the machine at the first instruction in each stream. That is, the first instruction encountered which generates a miss constitutes the beginning of the next stream, which means that the previous instruction is the end of the preceding stream.

Streams of the same length (in terms of number of instructions) are placed into buckets. The probability that a randomly selected instruction stream will be from a given bucket is then computed. If the CIPD is represented by the function $\Psi(L)$ where $L$ is an instruction length, $\Psi(L)$ will return the probability that an instruction stream of length greater than or equal to $L$ will be encountered. Thus, for any program, $\Psi(0) = 1$, and if $\gamma$ represents the maximum length of any instruction stream, $\Psi(\gamma + 1) = 0$. Each of the CIPD graphs which follow represent exactly the function $\Psi(L)$ for each experiment. $\Psi$ can also be used to determine the probability that an instruction stream of length less than or equal to $L$ will be generated. This function, called $\Psi^*(L) = 1 - \Psi(L)$.

It should be noted that the graphs are constructed from individual data points determined during program execution. Since the $\Psi$ always begins at 1 and eventually decays to 0, anything to the left of the beginning of the graph (usually $10^3$ instructions) will rapidly reach 1. Similarly, the "end-points" presented are not the true end-points (since they should always become 0); rather they represent the probabilities of the largest instruction streams encountered. Rather than presenting the entire function, these starting and ending points were chosen to better represent the graph and include more information.

There is no notion of weight contained within the CIPD, which can be thought of as "time spent executing." Instruction streams of very long length will show a relatively low CIPD, but could potentially represent the most significant percentage of the overall execution time.

## 6 Working Set Critical Mass

Of primary concern in the construction of PIM systems is the ability of a PIM to capture a significant working set to perform computation. Modern systems represent working sets in two ways: as a cache or as a page space.

### 6.1 Caches

Four cache configurations were examined in detail using PIMs of 1, 2, 4, 8, 16 and 32 MB. The configurations were a 256-bit block direct mapped cache, a 2k-bit block direct mapped cache, and 256-bit block 4-way and 8-way set associative caches. (The choice of block size corresponds to assumptions regarding convenient memory access discussed above.) For the purposes of these experiments, only heap data was analyzed (that is, code references were ignored under the assumption that code which is not self modifying can be duplicated across any number of nodes, and stack references were ignored as the size of the active area in the stack tends to be relatively small [18]).

Figure 3 show the typical cache result, in this case using the Method of Moments benchmark. As can be seen from the miss rate, increasing the cache size does not significantly impact the miss rate above cache sizes of 16 MB. Further more, for the most effective configurations (the 256 bit block and 2 k-bit block direct mapped), it does not effect it at all from the initial 1 MB size on. This indicates that temporal locality is exhausted for these benchmarks with a relatively small PIM size. (In this regard, the data management benchmark fared the best, however its best configurations did not improve above 4 MB PIM sizes.) Full simulation details can be found in [18].

Somewhat counter-intuitively, the set associative caches performed worse than the direct mapped configurations. However, given that the caches are so large (as are the block sizes), many sets in both of the set associative configurations remained unfilled. The low reuse of many of the benchmarks further accentuated this outcome.

The increased spatial locality provided by paged memory spaces significantly improved performance. The next section will demonstrate a 1-2 order of magnitude improvement in performance.

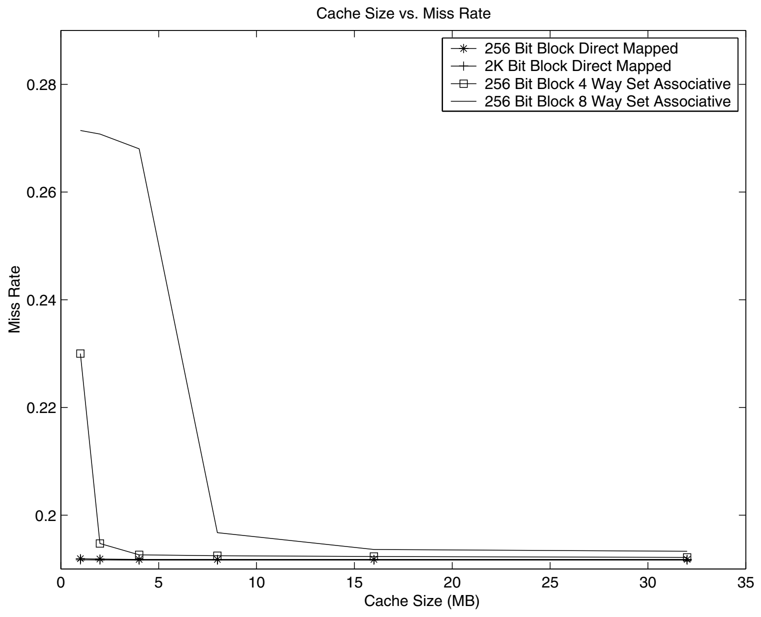


**Fig. 3.** DIS Method of Moments Cache Size vs. Miss Rate

## 6.2 Paged Memory

Programmers tend to allocate even pointer-based data in a relatively uniform fashion[23]. This accounts largely for the improvement in performance demonstrated by paged memory configurations. Of primary concern here is the degree to which larger pages are effective, given that on a PIM with a relatively small physical address space, pages which are too large may not allow enough windows into the address space to be effective.

Figure 4 from the DIS Data Management shows the miss rate versus the number of pages on a given node for pages of various sizes (4 KB to 256 KB). The key result given by this graph is that for all PIM sizes tested (1 MB to 32 MB) increasing the page size uniformly improved the miss rate. This indicates that in each case not only was the larger page able to provide additional spatial locality, but having fewer windows into the overall address space did not adversely affect the miss rate.

Obviously larger page sizes place a greater demand on the system's interconnection network during a miss. However, it should be noted that the type of system under examination is assumed to have a very high bandwidth interconnect (with a corresponding high latency for access). Additionally, due to the enormous number of nodes – potentially $O(10^6)$ – possible in such a system, it makes sense to place a greater premium on directory services and the simplicity of name translation than on page transmission time. Finally, no assumptions

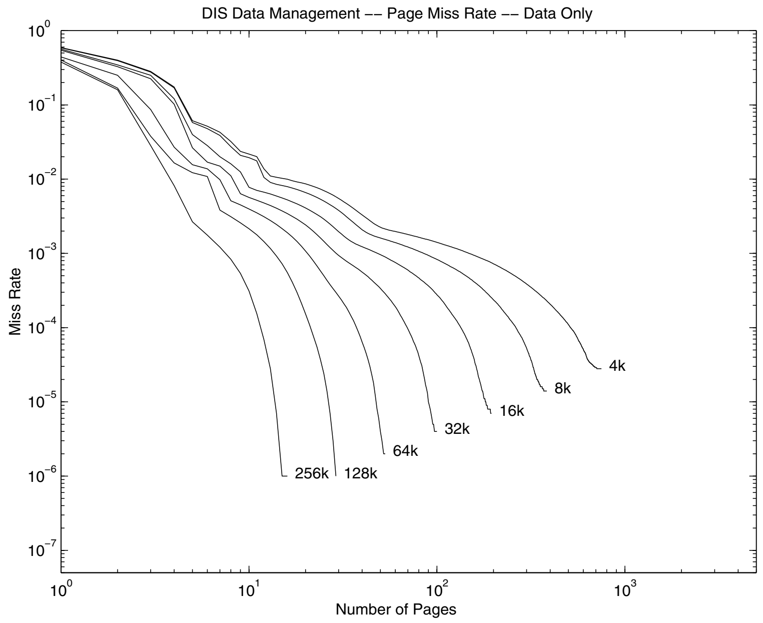


**Fig. 4.** DIS Data Management Overall Miss Rate

have been made in regard to the location of pages being retrieved (in another area of physical memory, on disk, in a COMA arrangement, etc.), thus assigning a "miss penalty" for the purposes of these experiments is largely irrelevant.

**Table 1.** Working Set CIPD Mean Values (256 KB pages)

| PGM | 2 MB | 4 MB | 8 MB | 16 MB | 32 MB |
|---|---|---|---|---|---|
| DM | 10K | 13.16M | 13.16M | 13.16M | 13.16M |
| FF | 200 | 300 | 600 | 1.03M | 1.89M |
| MoM | 5700 | 9.26M | 2.62M | 9.62M | 9.62M |
| IU | 632K | 655K | 9.26M | 9.26M | 9.26M |
| RAY | 117K | 202K | 1.11M | 1.11M | 1.11M |

Tables 1 and 2 show the mean and median values of the CIPD ($\Psi(L)$) for each of the benchmarks. They show that relatively small PIMs (4 MB to 8 MB) are highly effective in capturing a working set for most benchmarks. The FFT is a highly unusual case in that 16 MB PIMs are of particular strength in capturing the working set. This is not surprising, however, since the matrices involved are $O(15MB)$ in size.

**Table 2.** Working Set CIPD Median Values (256 KB pages)

| PGM | 2 MB | 4 MB | 8 MB | 16 MB | 32 MB |
|---|---|---|---|---|---|
| DM | 2K | 20.55M | 20.55M | 20.55M | 20.55M |
| FF | 2K | 2K | 2K | 1.54M | 1.72M |
| MoM | 18K | 393K | 393K | 393K | 393K |
| IU | 601K | 601K | 1.53M | 1.53M | 1.53M |
| RAY | 148K | 149K | 155K | 155K | 155K |

## 7 Mobile Threads

Thus far it has been shown that an individual PIM is capable of holding a significant workings set and that increasing the page size significantly improves run lengths on a given node. Furthermore, system design thus far has emphasized not only the long run lengths between remote accesses (due to the relatively low latency of a local memory access versus a remote memory access), but also simplicity in tracking the location of data. In extremely large systems, maintaining a directory of highly fragmented data becomes complex both due to synchronization and storage requirements[18].

Consequently, it becomes increasingly viable to move the computation instead of the data in a *mobile thread* environment. This system, similar to Active Messages[7], extends from the ability of a parcel to invoke computation on a remote node. Under this model, a thread executes until a remote access is generated. At that time, the location of the remote names is determined, and the thread is packaged into a parcel for transmission to the remote node. Upon receipt, the remote node continues the thread's execution.

There are several potential advantages to moving the computation:

– Page tables or other data structures managing the translation of names become small.
– Static data placement significantly reduces the synchronization involved in updating distributed versions of those structures.
– The physical location of a given computation need not be tracked at all. Threads can freely roam the system without causing the update of complicated, distributed data structures. Specifically, if various threads communicate through shared memory, they need not know the physical node upon which the thread with which they are communicating resides, only the location of the shared memory.
– Programming models can emphasize moving to a given node, exhausting the data present, and moving on. Simple mechanisms for delivering such data can easily be provided by the runtime system.
– No round trip communication is necessary since the thread can move to the data rather than requesting data which must then be returned. This eliminated one high latency penalty upon each movement.

Naturally, there are potential problems with such an arrangement:

– Load balancing may be difficult, especially if data placement relies upon highly shared data structures (that is, a given node could become a bottleneck if sufficient computation resources are unavailable).
– The runtime system must be capable of dealing with threads which have run amuck.
– It may be impossible to group data such that related items are together. (This experimentation, using benchmarks which are among the worst known in this regard, indicates that this is really not a problem.)

It is impossible to address all of these problems in this paper, particularly since this experimentation is still in the preliminary phase. Furthermore, the single threaded model adopted for these experiments is incapable of examining contention amongst several mobile threads.

The current model does, however, allow for the characterization of memory access patterns generated by a single mobile thread. Since this single thread represents the main loop of the program, its memory demands should be no smaller than those of its children.

Given the potential difficulties of mobile thread execution, it is likely that a hybrid model will be adopted. For example, data which is heavily shared but not often modified could be duplicated amongst multiple nodes. Additionally, it should be noted that each of the potential problems listed above also occurs with systems which only move data.

## 7.1 Execution Model

Figure 5 shows the two potential types of mobile thread movement. In the first form, each time a remote memory access is generated a thread is packaged and moved to a new node. A slightly more complex model allows for the thread to communicate with the node upon which it was previously executing in recognition of the fact that some data from that node is probably still necessary during the computation. (This data can, in fact, be captured before the thread moves, which alleviates the reverse communication.)

Data contained on the previous node, if available, is tracked as a "look-back reference." This represents, ideally, what could be packaged up with the thread when it is moved so as to facilitate longer computation on the next node without communication. Of particular interest is the number of unique references to the previous node. Knowing this allows for the construction of data structures to effectively capture such references, and provides a measure of feasibility for mobile threads.

## 7.2 Data Layout

The experiments to be presented here allow for an extremely simplistic data layout. Heap data is divided into chunks equivalent to a given PIM size, and is held in place. Experimentation in [18] shows that the size of the active stack and

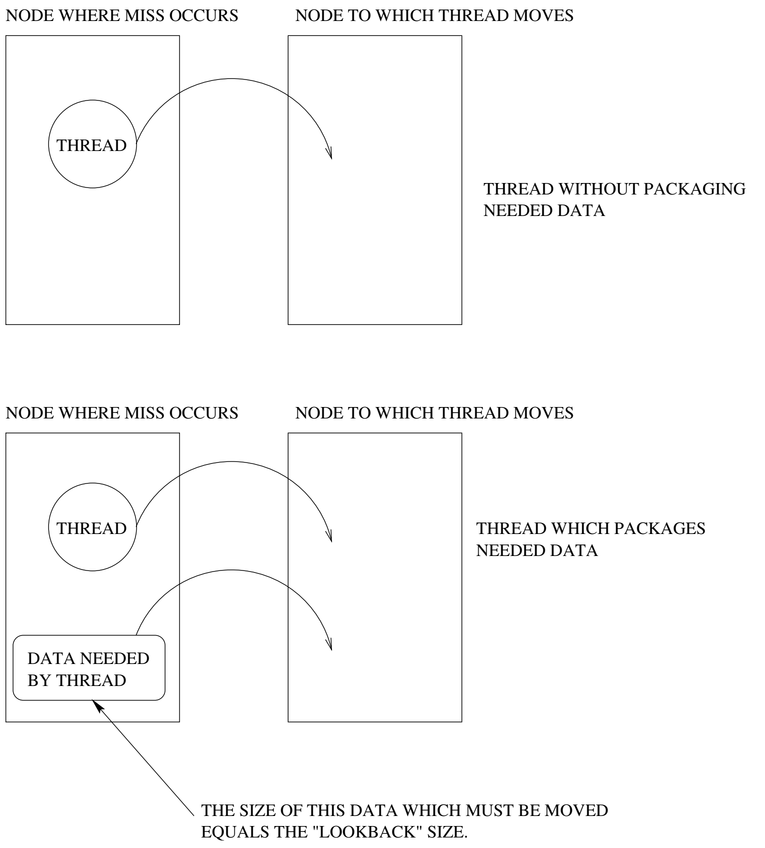

NODE WHERE MISS OCCURS
NODE TO WHICH THREAD MOVES
THREAD
THREAD WITHOUT PACKAGING NEEDED DATA
NODE WHERE MISS OCCURS
NODE TO WHICH THREAD MOVES
THREAD
THREAD WHICH PACKAGES NEEDED DATA
DATA NEEDED BY THREAD
THE SIZE OF THIS DATA WHICH MUST BE MOVED EQUALS THE "LOOKBACK" SIZE.

**Fig. 5.** Types of Thread Movement

code sections is relatively small (with will over 99% of each being served by in 32 KB of information or less).

The Spartan nature of this data placement tends to yield worst case results. It allows for no compiler, run-time, or user intervention in the policy for placement. Data is merely divided according to PIM size and placed accordingly.

## 7.3 Run Length Experimentation

Figure 6 shows the impact of backwards references on run length and the overall effectiveness of potential mobile thread computation. In this particular case (DIS Data Management) the results are easiest to understand (and are fairly typical). Because the data structure being traversed is a tree, the PIM size does not significantly alter the run-length data. (The index tree is significantly larger than even the largest PIM studied, therefore in eliminating half of the tree, the thread is required to go to a different node regardless of PIM size.)

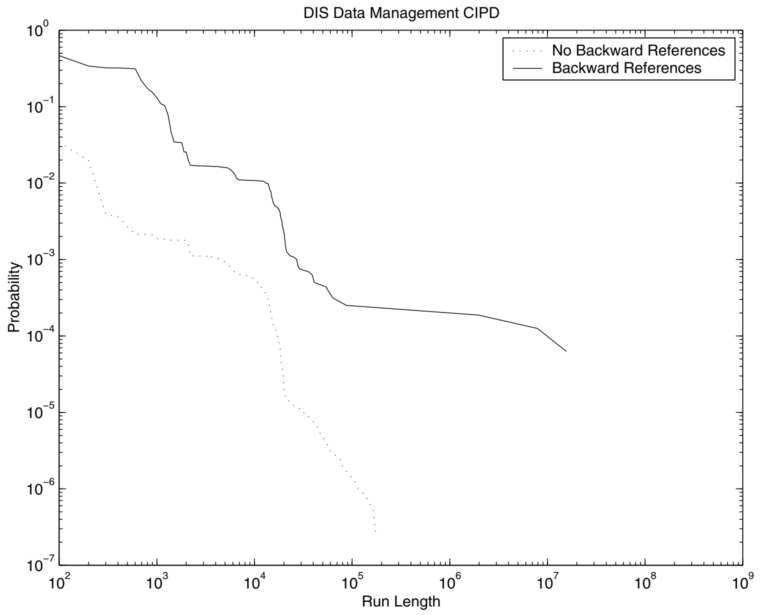


**Fig. 6.** DIS Data Management Run Length Results (CIPD, $\Psi(L)$)

Allowing for look-back references increased the maximum run length by approximately two orders of magnitude. Similarly, it increased the probability of executing a longer run by nearly an order of magnitude.

Figure 7 shows the most dramatic results. For the Method of Moments code, the maximum run length improved by over four orders of magnitude. Furthermore, not allowing for look-back references yielded particularly bad results – run lengths of over 1,000 instructions occurred less than 0.0001% of the time.

The numerous short run lengths in this benchmark can be attributed to the simplicity of the data placement scheme as related data structures are allocated with very low locality. (Specifically, several big matrices are allocated one after the other, and therefore reside on separate nodes.)

## 7.4 Look-Back Reference Results

Figures 8 and 9 show the probability density of a unique number of references to the previous node being made for a given instruction stream. In every run, except the Image Understanding benchmark, only 10 percent of instruction streams reference more than 10 unique 32-bit words from the previous node, indicating that a very small amount of data is needed to augment a thread once it has moved.

Figure 10 shows the worst case results given by the Image Understanding benchmark. The IU benchmark tended to thrash between the image it was look-

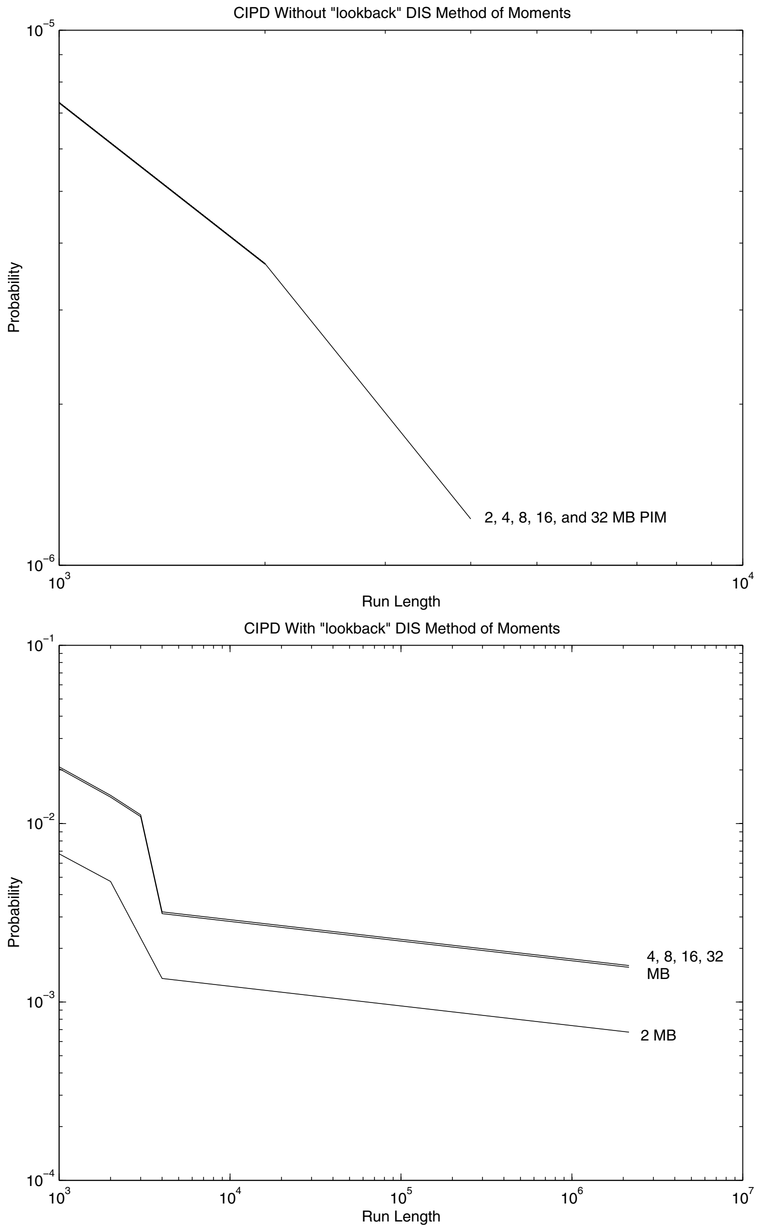


**Fig. 7.** DIS Method of Moments Results (CIPD, $\Psi(L)$)

ing for and that which it was examining. This problem can be alleviated by better data placement.

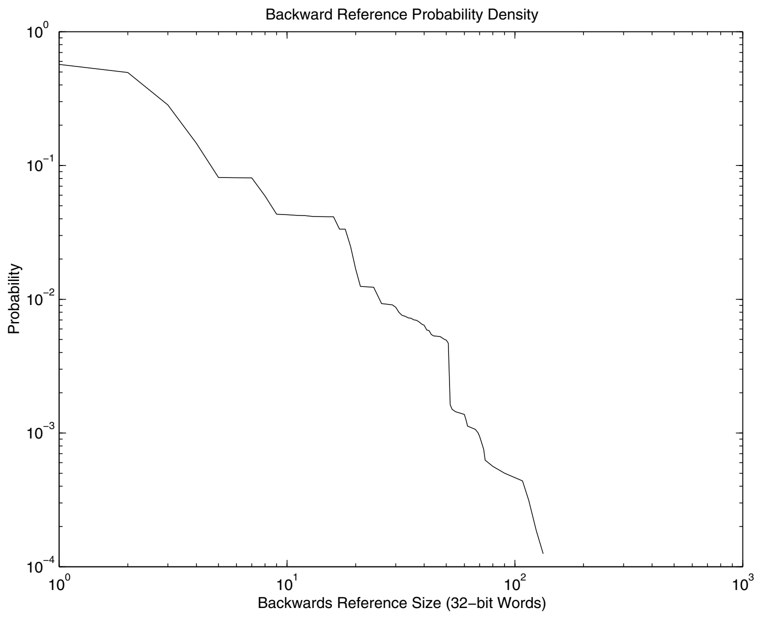


**Fig. 8.** DIS Data Management Look Back Reference Results

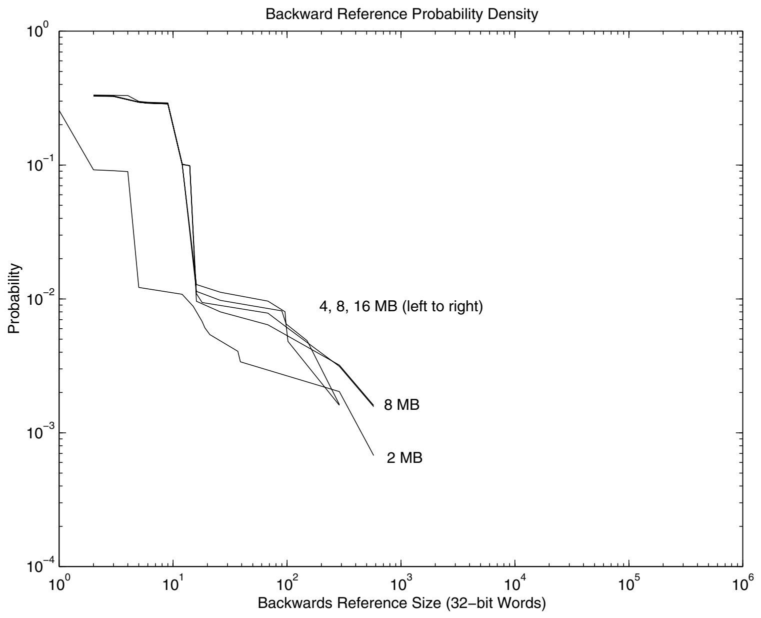


**Fig. 9.** DIS Method of Moments Look back Reference Results

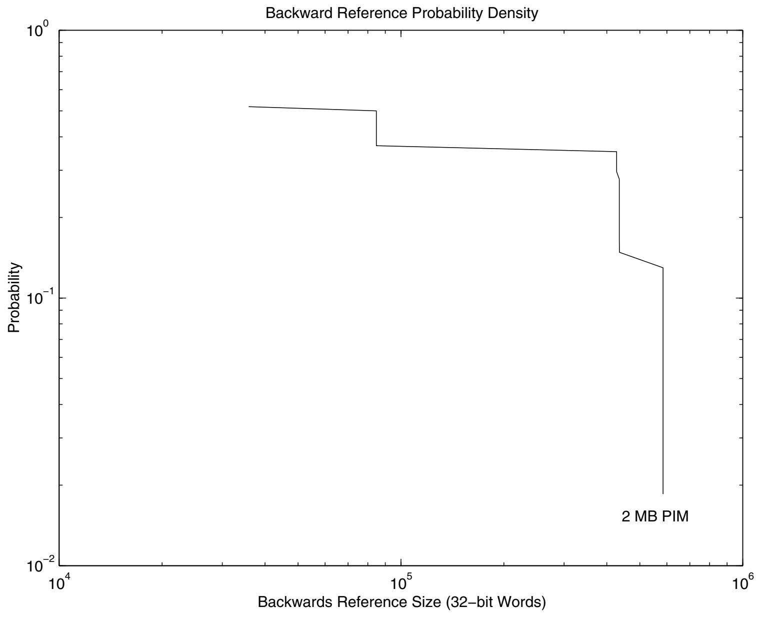


**Fig. 10.** DIS Image Understanding Look back Reference Results

## 8 Conclusions and Future Work

This paper examined the architectural parameters which effect program execution on PIM arrays using the Data Intensive Benchmark suite. Because these benchmarks exhibit complex, non-uniform memory request patterns, understanding their characteristics provides an ideal test-bed to flush out the architectural parameters necessary to take advantage of extremely low latency on-node memory accesses. Furthermore, by focusing on applications which have proven themselves difficult for typical memory systems to accommodate, this paper provides a set of “worst (realistic) case” memory access scenarios.

The paramount engineering problem upon which this work centered was the determination of the physical parameters of the design of the memory system (particularly how much physical memory a given node would need to sustain significant computation, and how that memory can be logically organized). While larger memories generally improved performance, it was shown that a relatively small PIM (with a 2 to 8 MB memory macro, for example) can sustain significant computation, and that, in fact, significantly larger PIMs were needed before another order of magnitude increase in executions between misses occurred.

Surprisingly, the increased potential to exploit spatial locality provided by large pages provided significant benefit in all the experimentation. Given that the benchmarks exhibit highly non-linear stride during memory accesses (due to pointer chasing or non-uniform matrix access), and each contained very large data sets, this result, in which fewer windows into the address space are available, took the experimentation in a different direction. Specifically, the number of windows into the address space on each node was reduced to the minimum (one) and the computation was allowed to move between nodes.

Generally, even given the simplistic data placement model, this mechanism proved effective, especially when coupled with the ability to "look back" at the previous node for data which may still be needed. After moving, the amount of data used on the previous node tended to be quite small (on the order of hundreds of bytes), implying that it can be effectively captured and packaged before the thread is moved.

Future work in this area centers upon refining the mobile thread model. A simulator capable of tracking multi-threaded versions of the DIS suite is currently examining the issues of contention, scheduling and traffic. Furthermore, work to define the data structures and hardware necessary to effectively capture look-back references and accelerate the packaging, as well as efforts to define multi-threading constructs capable of supporting inexpensive thread invocations and context switches.

## References

1. Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera System.
2. Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.
3. Atlantic Aerospace Electronics Corporation. *Data Intensive Systems Benchmark Suite,* `http://www.aaec.com/projectweb/dis/`, July 1999.
4. Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A New Memory Semantics for Massively Parallel Computing. In *ICS*, 1999.
5. Doug Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.
6. Michael J. Carey, David J Dewitt, and Jeffery F. Naughton. The OO7 Benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, 1993.
7. David Culler, Kim Keeton, Cedric Krumbein, Lok Tin Liu, Alan Mainwaring, Rich Martin, Steve Rodrigues, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. February 1995.
8. B Dembart and E.L. Yip. A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels, December 1994.
9. M.A. Epton and B Dembart. Low Frequency Multipole Translation for the Helmholtz Equation, August 1994.
10. M.A. Epton and B Dembart. Multipole Translation Theory for the 3-d Laplace and Helmholtz Equations. *SIAM Journal of Scientific Computing*, 16(4), July 1995.
11. Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOID*, June 1984.
12. J. M. Haile. *Molecular Dynamics Simulation : Elementary Methods.* John Wiley & Sons, 1997.

13. Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.
14. Peter M. Kogge, Jay B. Brockman, and Vincent Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain*, June 27-28, 1998.
15. Peter M. Kogge, Jay B. Brockman, and Vincent W. Freeh. PIM Architectures to Support Petaflops Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.
16. Banks Kornacker. High-Concurrency Locking in R-Tree. In *Proceedings of 21st International Conference on Very Large Data Bases*, September 1995.
17. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. *ISCA*, June 2000.
18. Richard C. Murphy. *Design Parameters for Distributed PIM Memory Thesis.* MS CSE Thesis, University of Notre Dame, April 2000.
19. Notre Dame PIM Development Group. *ASAP Principles of Operation*, February 2000.
20. SPEC Open Systems Steering Committee. SPEC Run and Reporting Rules for CPU95 Suites. September 11, 1994.
21. David Patterson, Thomans Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.
22. T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petaflops scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.
23. Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, Jim Dehnert, and Suneel Jain. Automatic Prefetching of Induction Pointers for Software Pipelining. *CAPSL Technical Memo, University of Deleware*, November 1999.
24. Sun Microsystems. *Introduction to Shade*, June 1997.

# Memory Management in a PIM-Based Architecture

Mary Hall and Craig Steele

University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292, USA
{mhall,steele}@isi.edu

**Abstract.** The DIVA (Data IntensiVe Architecture) system incorporates Processing-In-Memory (PIM) chips as smart-memory coprocessors to a host microprocessor. It exploits the inherently high on-chip memory bandwidth and additionally provides a separate memory-to-memory high-bandwidth interconnect across the system. By design, the DIVA system architecture targets a broad range of applications, including those with irregular data access patterns. At the same time, DIVA supports familiar programming paradigms from parallel computing and offers an evolutionary migration path for application development.

The DIVA project is constructing a demonstration system using a conventional superscalar host processor with a main memory composed of VLSI PIM chips in place of standard DRAMs. This system has a novel mix of operating-system challenges, combining aspects of conventional "dumb" memory management and both shared- and distributed-memory multiprocessor operations. This paper describes our solutions to the memory-management problems posed by this multifaceted environment.

## 1 Introduction

The Data IntensiVe Architecture (DIVA) project is building a workstation-class system using embedded-DRAM technology to replace the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. The goal of the project is to significantly reduce the ever-increasing processor-memory bandwidth bottleneck in conventional systems. System bandwidth limitations are thus overcome in three ways, as illustrated in Figure 1: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processor-memory nodes per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus.

This paper describes memory management in DIVA. Two aspects of the DIVA project distinguish its memory management requirements from that of other PIM-based architectures.

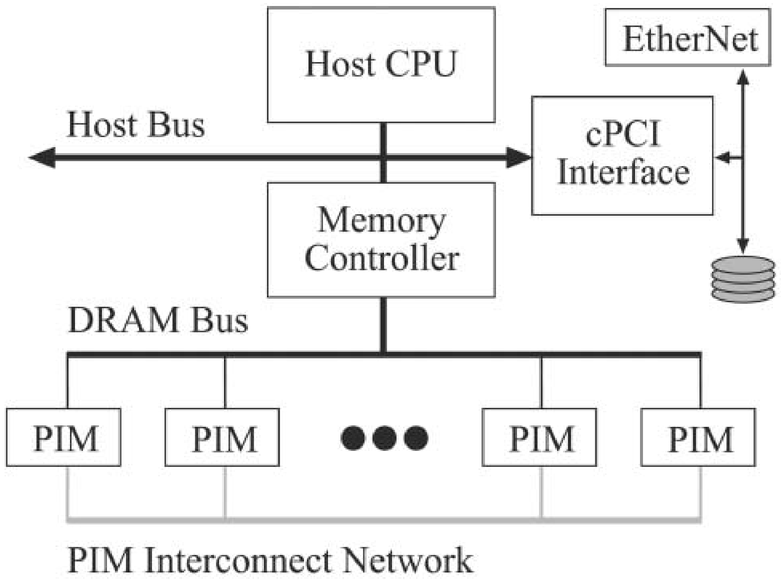


**Fig. 1.** DIVA System Architecture.

- The PIMs serve as the only memory for a standard host microprocessor assuming the dual roles of "smart memories" and conventional memory.
- DIVA targets applications that are most severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets.

As compared to system-on-a-chip solutions [6,5], and multiprocessors made up solely of PIM chips [7,4], DIVA's support for conventional memory accesses from an external host requires a dual view of memory, from the host perspective and the PIM's perspective. Other PIM architectures address this challenge by restricting PIM functionality to SIMD execution on large streams of data, at the host's direction [1,2]. In DIVA, we support a much broader range of programming paradigms, including task-level parallelism and in-memory accesses to pointer data structures. As a result, DIVA requires a memory model that supports independent threads of control and efficient translation in memory without necessitating host intervention.

A previous paper presents an overview of the DIVA project and describes a memory model to support these requirements [3]. This paper discusses the memory management support needed to realize this memory model.

## 2 Overview of Memory Model and Address Translation

The DIVA memory model attempts to satisfy a number of potentially conflicting requirements:

- scalability to tens of PIM chips;
- efficient hardware mechanisms amenable to straightforward implementation;
- an abstract machine comprehensible to both programmers and compiler writers;
- compatibility with conventional memory models and memory interfaces;
- support for virtual memory (i.e., paging to/from disk) and swapping; and,
- supporting correct functionality of both shared- and distributed-memory programming models.

The unifying concept for the DIVA memory model is communication via a global address space shared by the host processor and all the PIM node processors. Not all memory need be shared, however, so our hardware supports local PIM address spaces as well. All of the processors in our demonstration system use 32-bit addresses, but the model can be advantageously extended to future 64-bit systems.

To interpret addresses in PIM code and data, a PIM processor must support a translation mechanism. However, the space and time overhead of maintaining conventional page tables at each node is prohibitive. To simplify translation hardware, we classify DIVA memory according to usage:

- global memory is a single address space distributed across nodes, visible to applications running on the host and PIM nodes.
- dumb memory is a region of a node's memory allocated as conventional pages in a host application's virtual space and untouched by PIM node processing.
- local memory is a region of a node's memory used almost exclusively by PIM routines. Certain exceptional functions of the host operating system, such as initialization and context management, will also access this memory occasionally, requiring well-defined data-sharing conventions.

The physical memory on each PIM chip is flexibly partitioned into these three distinct uses. Dumb memory is managed exclusively by the host operating system in standard ways, with address translation handled solely by the host processor's memory-management hardware. Figure 2 depicts the two more interesting uses of PIM memory, as part of the shared global address space, or as PIM local memory. The DRAM memory associated with the global address space is physically distributed across all the PIM nodes involved in a computation. Addresses in the global virtual address space are consistent for the host and all PIM node processors, so that pointer-based data structures can be freely shared. In contrast, while the host has physical access to the PIM DRAM used as local memory, the host and PIM node processors will see it at different virtual addresses.

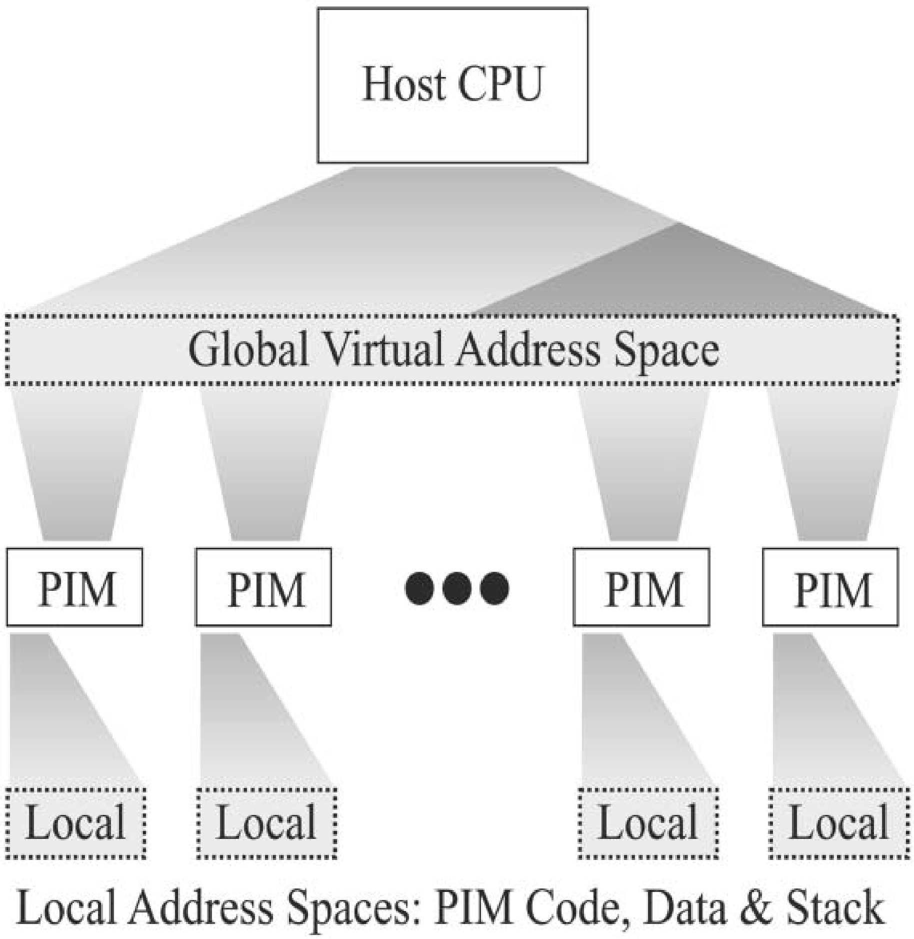

Host CPU
Global Virtual Address Space
PIM
PIM
PIM
PIM
Local
Local
Local
Local
Local Address Spaces: PIM Code, Data & Stack

**Fig. 2.** Shared global segments, unshared local segments.

The host processor can access PIM memory via its memory bus. To avoid saturation of this bus, PIM-to-PIM communications occur primarily by means of a distinct high-bandwidth network between PIM chips. The hardware directly supports shared-memory operations between the host and PIM memories, but PIM-to-PIM communications are implemented by network communications in the form of parcels (Section 2.3). Parcel operations are hardware assisted, but require software processing by either user- or supervisor-level code at both ends. Efficient network interface and interrupt mechanisms have been developed to support parcel functions.

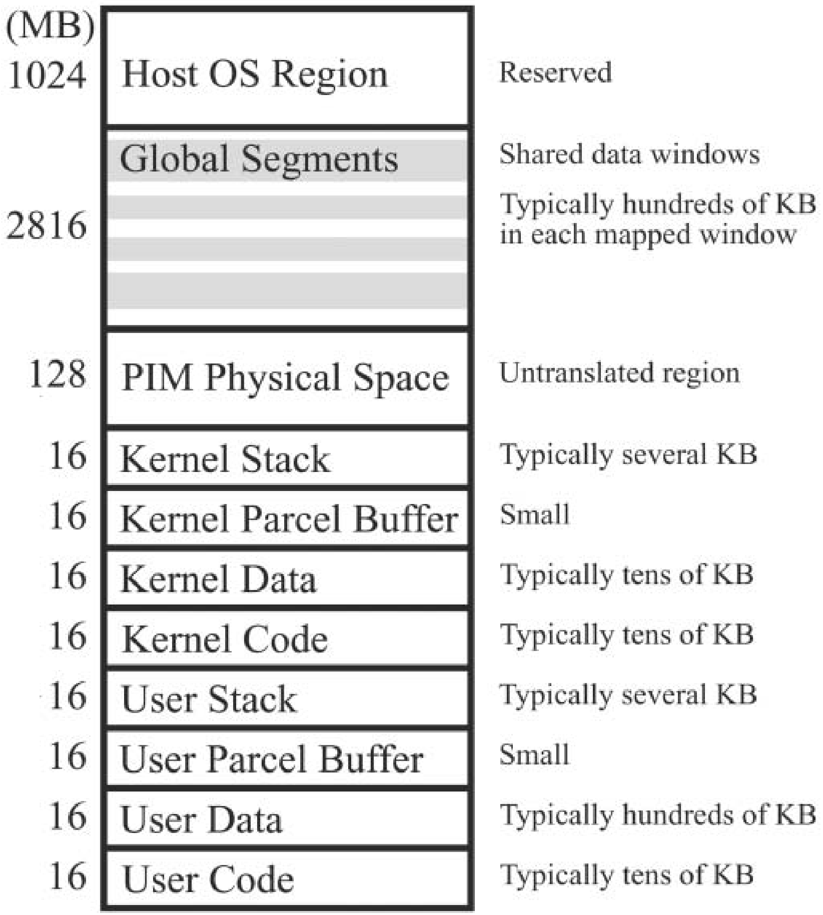


**Fig. 3.** PIM node processor address map.

## 2.1 Address Translation for Locally Mapped Data

A node must be able to rapidly determine if an address is located in its own memory, and if so, find the physical address. Each node therefore maintains translations for virtual addresses currently residing on it, including local memory and its portion of global memory. To condense translation information, we use segments, each of which is defined by segment registers containing a physical base address and limit. The segments are described by the PIM address map shown in Figure 3. The local memory region is partitioned into eight segments at fixed virtual bases, for kernel code, stack and data, user code and data/stack, and for kernel and user network-communication buffers. A small number of global segment registers are also used; since global segments must be able to map portions of a shared virtual address space much larger than the physical memory of an individual node, global segments must be represented by both a virtual and physical base address register.

Figure 3 shows the virtual address map for a PIM node processor. The virtual size of each region of the map is shown on the left; typically only a fraction of that virtual address space will be used, as noted on the right of the diagram. The lowest-addressed (bottommost) segments of the address map define the local-memory user-mode space for the current process. The next group of segments defines the local-memory supervisor-mode space for the kernel, which is the same for all processes. The kernel can also access the PIM's DRAM without address translation via the physical space region.

Each PIM has a small number of relocation registers to allow it to map portions of the shared global address space to the node's physical memory. The aggregate size of these "windows" into the shared address space is limited by the amount of physical memory available on a node. The top region of the map is unused, but reserved to conform with the host operating system's address map.

## 2.2 Translating Remote Addresses

Access to parts of the global address space not mapped to physical memory on the node is possible via the network, but less efficient than a mapped access. A parcel must be sent to the node which contains the physical memory to be accessed, and a response parcel received and processed, either by user or kernel code.

DIVA determines the location of remote data via a two-stage process. The virtual address of a datum is hashed by hardware to determine the "home node" of the datum [8]. The home node may or may not be the present physical location of the datum, but serves as the centralized directory and manager for it. The home node will either perform the operation itself, if the datum is resident, or forward the request to another node, if the datum resides elsewhere.

Therefore, a node must maintain translation information for only eight local segments plus a small number of segments for its portion of the global memory, as well as for any global data for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

### 2.3 Parcels

All PIM-to-PIM network communications are performed by sending and receiving messages in the form of parcels. Parcels are an object-based variant of active messages [9], distinguished from active messages in that the destination of a parcel is an object in memory, not a specific node. From a programmer's view, parcels, together with the global address space supported in DIVA, provide a compromise between the ease of programming a shared-memory system and the architectural simplicity of pure message passing. Remote operations or accesses can be accomplished through parcel sends and receives; application programs only need specify the address of an object, and not the processor upon which the object resides.

Structurally, a parcel packet has a 256-bit payload and 96-bit header, which includes:

- the memory address of the target object, expressed in the application's address space,
- the environment id (eid) of the process in the host that is executing the application,
- a command identifying the function to be performed by the node associated with the target when the parcel arrives.

The 256-bit payload serves as arguments to the command. A parcel requiring more bits must be sent in multiple packets. The payload size matches the PIM node data bus width; streaming packets may be sent in a single bus cycle. The network interface supports both user- and supervisor-mode access to parcel sending and receiving hardware via the user and kernel parcel buffer segments. User-mode parcel processing is more efficient but less robust than kernel-mediated operations, so will typically be restricted to compiler-generated code or library routines. Error conditions cause invocation of either user- or supervisor-mode handlers.

## 3 Overview of Memory Management

Memory management functions are divided between two types of kernels in the DIVA system. On the host processor, the standard operating system (in DIVA, Linux) is augmented with functionality to support PIMs. On each PIM processor, there is a tiny run-time kernel that is always resident. A primary responsibility of the PIM run-time kernel is to manage parcel communication between PIMs [3]. The run-time kernel performs buffer management of incoming parcels, and directs context switches between different threads in the same user program, or between user program and kernel. The run-time kernel also performs required software intervention in response to interrupts and exceptions on the PIM processor. In addition to these autonomous functions of the PIM run-time kernel, it also must collaborate with the host on system-level operations, such as loading PIM programs and data, memory management of PIM-visible segments, and PIM context switches between different user programs (note that most host context switches will not involve the PIMs).

This division of labor is motivated by the dual goals of keeping the PIM run-time kernel as small as possible, and making only moderate changes to the standard function of the host operating system. Unlike standard multiprocessor systems, the host, which has a system-level view, remains a central figure in system-level scheduling, disk I/O operations, and memory management. The challenge in this collaboration between host and PIM system software is that there are really two views of memory that must be maintained. For dumb pages and for disk I/O of PIM-visible segments, the host sees memory as standard 4Kbyte pages; the PIM run-time kernel instead views PIM-visible memory as variable-sized segments. Reconciling these two views through different system functions is the subject of the remainder of this paper.

## 4 Memory Allocation: Virtual vs. Physical

The portion of memory used by the host as dumb memory is managed by the host operating system using standard allocation, paging and swapping mechanisms. The memory devoted to PIM local memory, and global shared memory, must be managed via a collaboration between host and PIMs. The most unusual aspect of this collaboration is memory allocation.

Figure 4 shows the functions associated with memory allocation, and whether they are performed by host or PIM. There are three phases to allocation: (1) host allocation of contiguous virtual address spaces for global and PIM local segments using the *Reserve* functions; (2) physical allocation of an object and binding to reserved virtual segments; and, (3) mapping of existing global objects to a global segment for sharing between PIMs. Deallocation (*GlobalFree*) frees physical memory but does not shrink the virtual-space allocation.

The standard memory allocation functions *malloc* and *free* can be used on either the host or PIMs; the meaning depends on where the functions are executed. On the host, a call to *malloc* performs a standard allocation from dumb memory. On the PIMs, it allocates memory from the PIM's local heap segment. Memory obtained from *malloc* is private to a process and unsharable.

### 4.1 Virtual Memory Allocation

Using a segmented approach requires that data in a segment reside in contiguous virtual addresses. For this reason, as part of the allocation process, we must reserve a contiguous chunk of the virtual address space for each segment prior to physical allocation. The virtual memory allocation is performed by the host using the *Reserve* functions for global and local segments. Because the virtual address space is quite large, these reservations should always strive to overestimate the space requirements of the segment, particularly since growing a segment beyond what was initially reserved results in very costly adjustments in virtual and physical allocations. Linux supports this reservation process by clustering free

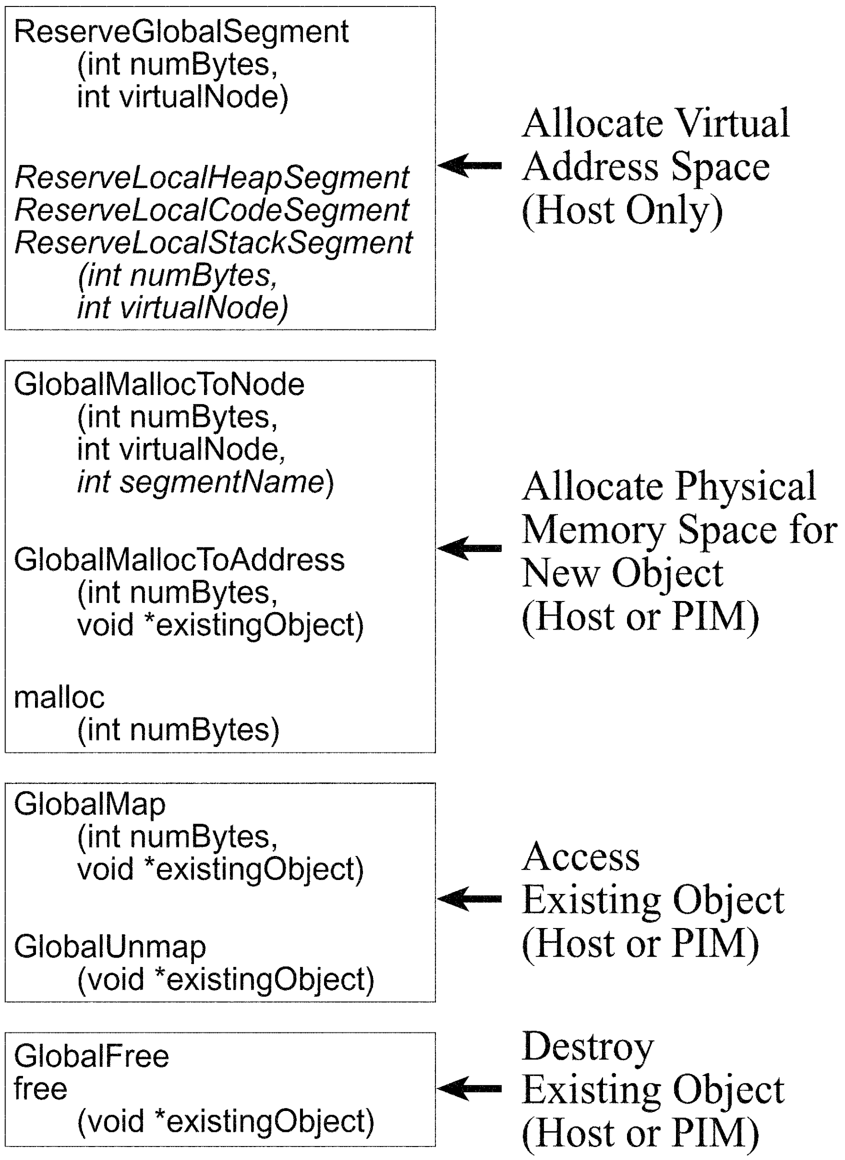


**Fig. 4.** Host and PIM memory management functions and steps of memory allocation.

pages in the virtual address space together; reservations select a cluster that matches the requested size.

Multiple global segments can be reserved by separate invocations to the *ReserveGlobalSegment* function. Each global segment reservation will create a new segment with a unique name (see Section 7.2). The segment name can subsequently be used optionally by allocation functions, as discussed below. Similar functions exist to allocate the virtual address space for PIM-local code, data and stack segments. These are italicized to indicate that they are optional, since standard default values often suffice.

## 4.2 Physical Memory Allocation

The physical allocation is performed through a collaboration between host and PIM. As part of physical allocation, the page table entries on the host are filled. On the PIM side, segment registers may be updated.

The functions shown in Figure 4 allocate a specific object to a segment. However, global and per-node local memory allocation and deallocation could swamp the host operating system with fine-grained memory allocation requests. Behind the scenes, we distribute this task using a two-level scheme where coarser-grained requests to the host are made by each PIM run-time kernel to replenish locally managed memory pools of pre-allocated global and local memory. This approach keeps the host involved in memory allocation, but still permits the PIMs to allocate memory independently as needed for managing pointer-based and other dynamic structures during PIM computation.

The DIVA programming model offers a globally addressable, distributed address space on shared data. PIM applications perform correctly when accessing non-local memory, either by communicating via the parcel mechanism, or by retrieving data in response to a more expensive address translation fault. Nevertheless, just as with distributed-shared-memory architectures, to achieve the best performance, an application must whenever possible co-locate data with the computation that accesses it. For this purpose, there are two flavors of memory allocation functions. The *GlobalMallocToNode* function associates allocated data to a specific virtual PIM home node. An optional *segmentName* argument permits this allocation to occur within a specific global segment. To allow two related objects to be collocated without requiring the virtual PIM identifier, the *GlobalMallocToAddress* function instead permits dynamic allocation of objects to the same virtual PIM node and global segment upon which another datum resides.

To simplify the programming model, *GlobalMalloc* functions performed on the PIM match the interface used on the host. Most of the time these functions will be used to allocate data from the PIM's locally resident global segments, but it is possible for a PIM to perform an allocation on a remote PIM node. The effect of such an allocation is to allocate virtual addresses from the remote node, and locally map the object to the requesting PIM's global segments and physical storage. Such an allocation can be performed to support efficient updates of the remote data prior to forwarding them to the remote node (see Section 7.3).

### 4.3 Mapping Existing Objects to PIM Global Segments

Like the *GlobalMalloc* to a remote node, it is sometimes desirable to temporarily map non-resident global data to facilitate sharing among PIMs. The *GlobalMap* function performs this mapping to global segment registers, and *GlobalUnMap* returns the data to its home node (see Section 7.3).

## 5 Paging

To perform computations requiring access to global data structures larger than the actual amount of physical memory, we support a virtual-memory "paging" mechanism for PIM-process memory. (We use the slightly inaccurate term "paging" in preference to the technically preferable term "global-segment access fault management" for brevity.) If the memory access which caused the fault references a datum resident in a PIM memory, it can be resolved without troubling the host operating system, by retrieving the accessed datum via a parcel request to its home node. On the other hand, if the home node returns a message indicating that the requested datum is not resident in the system's physical memory, the initiating PIM kernel must request paging service from the host, which is connected to the disk backing store.

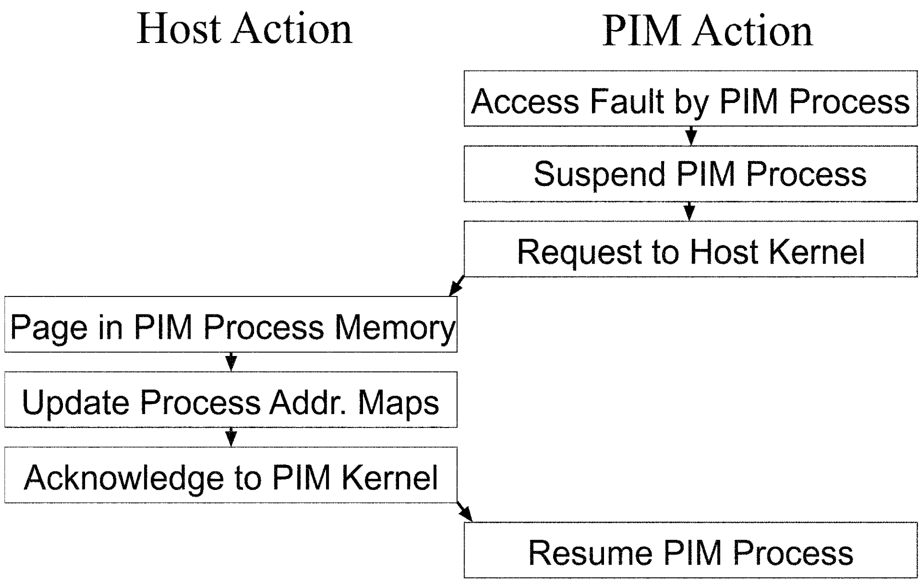

Host Action
PIM Action
Access Fault by PIM Process
Suspend PIM Process
Request to Host Kernel
Page in PIM Process Memory
Update Process Addr. Maps
Acknowledge to PIM Kernel
Resume PIM Process

**Fig. 5.** Paging sequence.

As shown in Figure 5, the faulting PIM process is suspended until the datum is paged in from disk, and the host kernel becomes the owner of that process context during the memory reorganization. The host pages in a section of the global virtual space containing the datum. The paged-in section is typically mapped to a distinct segment of the global space. However, if the faulting address is adjacent to an existing locally mapped segment, the segment may be extended to contain it.

After the host kernel has resolved the fault and adjusted the faulting process' PIM context mappings, it returns ownership of the context to the PIM kernel, which reloads the PIM address translation hardware when it reactivates the process.

The paging system is a useful but relatively expensive feature, best used sparingly.

## 6 Contexts and Swapping

A DIVA PIM node supports very efficient context switching for the most common cases, either switching between a user program and the PIM run-time kernel, or between two distinct threads within the same user program. Switching to the run-time kernel requires no change to segment registers, and requires minimal saving and restoring of register state. Switching between different threads in the same user program, such as when performing the command associated with an incoming parcel, requires modification to only two of the segment registers, but does require saving and restoring of portions of the register state. In either case, there is no need to swap memory in or out in response to a context switch.

In performing its normal job scheduling function, the host may direct the PIM node's context to change to a different user program that requires PIM functionality. In this case, a full context switch is necessary, saving all the register state as well as updating the program-specific segment registers. Further, memory may need to be swapped in or out. If the user-code physical memory is swapped out and recycled, the content of the PIM node processor's instruction cache must also be invalidated by software, since the new program's code memory may overlap with the previous program's. (Our processor, like many others, does not enforce coherency in the instruction cache hardware.) Note that at any time, the host may be executing in a different context from one or all PIMs; for most host context switches, it will not be necessary to change the PIM node context.

The host operating system is responsible for creating contexts for the PIMs, and also for updating contexts in response to major system context switches. (Lightweight PIM context switches, e.g., multithreading, do not involve the host.) To facilitate host management of contexts, during initialization the host creates a data structure, mapped to the PIM's memory, that it shares with the PIM run-time kernel. While it is possible for the host to build this data structure through a series of parcels sent to the PIM run-time kernel, for efficiency we permit the host operating system in privileged mode to write directly into portions of the PIM run-time kernel data segment. The host also updates this structure in response to a system context switch.

### 6.1 Contents of Context

Figure 6 graphically depicts the contents of a context. On initialization of a user program, the host performs virtual memory allocation of the segments, as discussed in Section 4.1, and writes the range of allocated virtual addresses into the context data structure in the PIM run-time kernel segment. The remainder of the context is reset to default values. As a result of physical memory allocations, the physical segment mappings are added to this structure. The remainder of the fields are filled in by the PIM run-time kernel when saving state as a result of a context switch. When this context is restored, the host updates the segment mappings as needed.

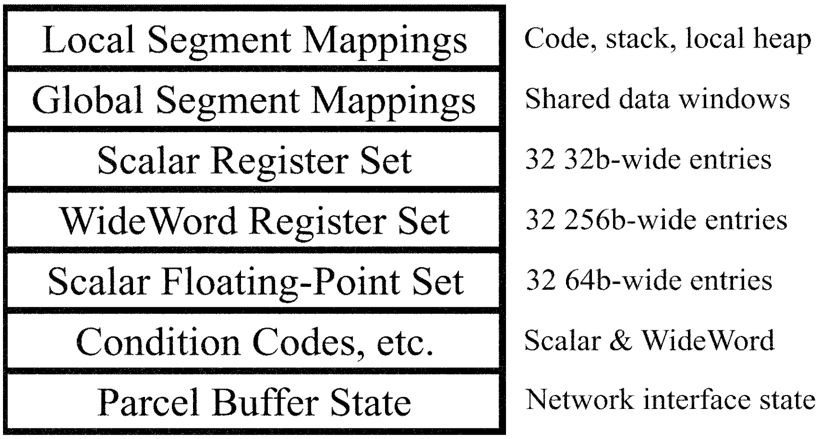


**Fig. 6.** Contents of context.

### 6.2 Swapping

Swapping is another mechanism for supporting computations with large memory requirements. Many computations can be broken up into distinct phases which need not be simultaneously active. Peak memory requirements may be reduced by swapping out inactive processes or low priority active processes. Swapping is somewhat similar to paging; the primary distinctions are that the entire context is moved to the disk backing store, freeing all the process memory, and that the host operating system, rather than the PIM kernel, initiates the swap as part of its overall scheduling function.

The sequence of actions required to effect a process swap and restore is sketched in Figure 7. As in the paging sequence, the ownership of the process context and its associated resources passes from the PIM kernel to the host operating system when the PIM process is suspended. Restoring a process context from an out-swapped state is quite similar to the initial instantiation of the process.

Swapping out a context frees most local process resources for reuse, but does not free memory used to store global segments, since they are likely to be in use by related processes on other PIM nodes. The global memory use may well exceed the local, so this is a potentially major problem for our storage reclamation capabilities. To be able to effectively manage groups of related processes, and to be able to decide when their associated global segments may be swapped out, we adopt a system of naming global segments, discussed in Section 7.2. We can thereby "gang schedule" related processes which use particular global segments and swap out their local and global resources together. We can record process references to a given global segment by explicit segment mappings and by remote parcel accesses. Statistics such as these can be recorded by each PIM node kernel and stored locally. The host operating system will only need to examine and aggregate these distributed runtime statistics in the event of a requirement for a major swapping operation, such as phase transition for a very large computation.

In general, the host must attend to global changes in the PIM-based computation, *i.e.*, scheduling functions, where resource allocation policies may be altered, and to chores which require access to external devices, such as swapping or paging to disk. We minimize the host's workload, and its potential for saturation, by requiring it to perform only those tasks which are global in their essence.

## 7 Local and Global segments

Section 4 described how local and global segments are allocated; here we consider how they are managed. Local segments should remain fairly small, so there is little concern that portions of them will be paged to disk during active PIM execution. Rather, we assume that most of the data read or written by PIM computations will reside in global segments.

Global segments provide a mechanism for sharing global data between host and PIM or across PIMs. Data-intensive applications will have a large amount of global data that can easily exceed the available physical memory capacity; thus, it is desirable to break up global data into multiple global segments. Global segments can be much larger than the 4Kbyte page size of the system, and there can be many more global segments associated with a user program than are mapped to the small set of global segment registers on each PIM. As a result, data required by a portion of the computation of the PIM program may be spread across multiple global segments; to avoid thrashing, care must be taken to map these segments to physical memory simultaneously during this portion of the computation. The remainder of this section describes the mechanisms for organizing and managing data in multiple or very large global segments.

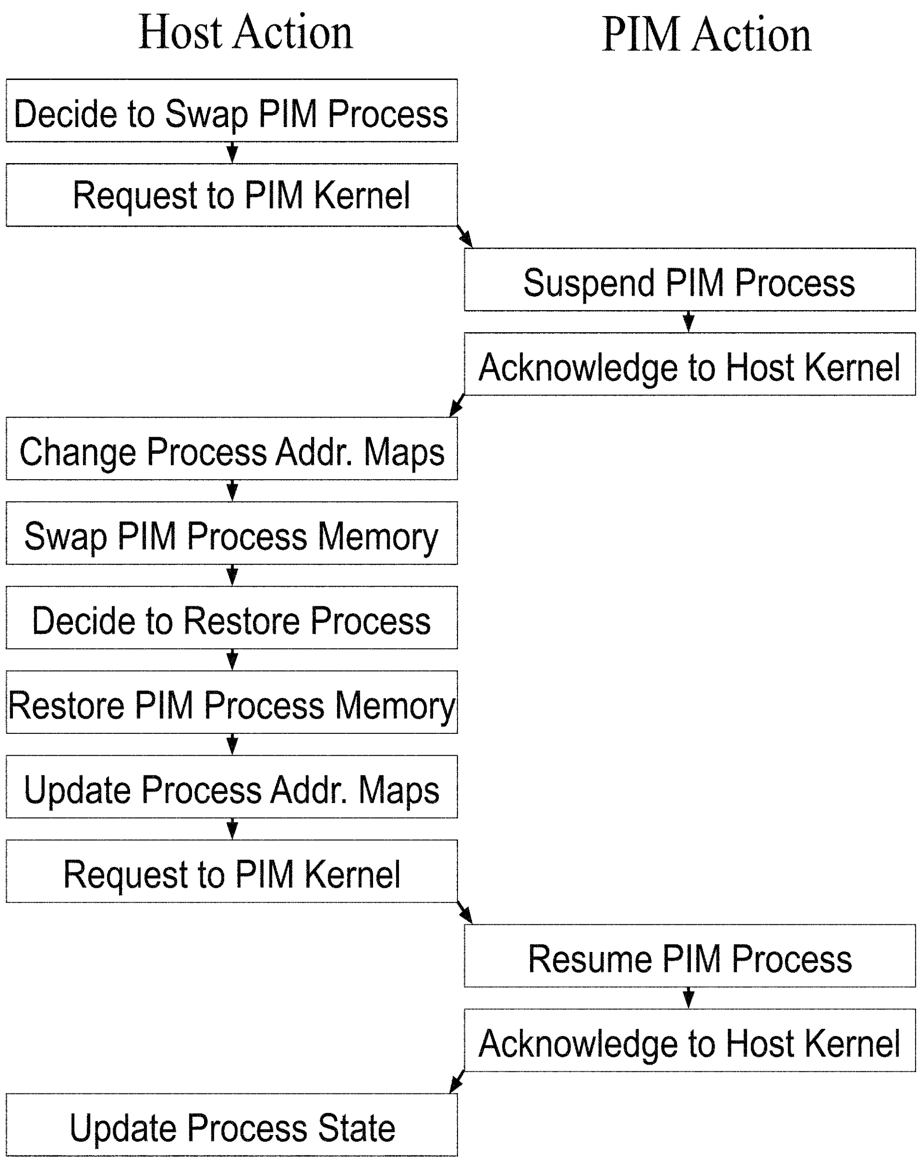

Host Action
PIM Action
Decide to Swap PIM Process
Request to PIM Kernel
Suspend PIM Process
Acknowledge to Host Kernel
Change Process Addr. Maps
Swap PIM Process Memory
Decide to Restore Process
Restore PIM Process Memory
Update Process Addr. Maps
Request to PIM Kernel
Resume PIM Process
Acknowledge to Host Kernel
Update Process State

**Fig. 7.** Swapping sequence.

## 7.1 Large Global Segments

In the absence of compiler or application-level support for defining segments, the operating system's default behavior is to create one or a small number of possibly very large global segments for a user application program. In this case, a single segment can be much larger than the available physical memory capacity, so that only a portion of the segment can reside in memory at a time.

Since each PIM has multiple global segment registers, even a single global segment can be managed as multiple segments by having distinct segment registers mapping different portions of the segment. This approach works well, for example, if an application is streaming through its data set sequentially. As it completes its accesses to data represented by one segment register, it can move on to data represented by another segment register. The operating system and PIM run-time kernel can page out the data associated with the former segment registers, and reclaim the segment registers and physical memory for a subsequent portion of the segment.

## 7.2 Assigning Names to Global Segments

While a single large segment can be managed effectively for streaming applications, in general, a more flexible mechanism is required for organizing data into multiple segments. For example, an application may revisit data in different phases of a computation; or, one data structure may be needed at the same time as another data structure in one phase of computation, and also required at the same time as a third data structure during a later phase of computation.

Our approach is to assign names to segments as they are being created, and permit the compiler or application program to optionally reference these segment names in memory allocation functions. For example, the effect of the allocation function *GlobalMallocToNode(int numBytes, int virtualNode, int segmentName)* is to allocate *numBytes* from the named segment *segmentName* on virtual PIM node *virtualNode*. (The effect of a *GlobalMallocToAddress* call is to perform the allocation on the same virtual PIM node and in the same global segment as that of the specified address.) By allocating two objects from the same global segment that are always used together, we can maximize the likelihood they will always be simultaneously in memory whenever they are being accessed. In cases where grouping all related data would result in too large a segment, the related data must be broken into multiple smaller segments, such that their size more manageably maps to physical memory, but at the same time, there are sufficiently few related segments so that all can simultaneously map to the small number of global segment registers on each PIM.

## 7.3 Sharing Global Segments across PIMs

As noted above (Section 4.2 and Section 4.3), mapping a global segment to local physical memory provides a mechanism for efficient sharing of large blocks of global data by asserting temporary ownership of a local copy of a data block that may be "homed" on another node. The home node of a datum in the global address space is a function of its virtual address, but the item may reside on another node. The home node provides a central access point for the item regardless of its actual location. In the absence of active mappings by other nodes, a datum will be (re)located to its home node.

The shared data block is created by either the host or a PIM node via the *GlobalMalloc* functions. The *GlobalMalloc* functions perform two distinct roles: allocating a block of physical memory and mapping a portion of the (previously reserved) global virtual address space to that physical memory. The *GlobalMallocToNode* function associates allocated data with a specific virtual PIM home node. However, if the function is invoked by code on a given PIM node, the initial physical memory allocation is made on that PIM node, which need not be the home node. A virtual-memory allocation request is sent to the remote home node, which records the location of that data object and returns a range of allocated virtual addresses drawn from its virtual pool. The requestor node maps that virtual address range to the physical memory it has allocated from its own physical pool. The requestor process is then free to access its instantiated data object at will. The process may terminate its use of the data object by invoking either the *GlobalUnMap* or *GlobalFree* function. Calling *GlobalFree* unmaps the object and indicates that its physical storage may be recycled and its content destroyed. Calling *GlobalUnMap* merely unmaps the object from the current process and indicates that its content should persist. The object will be relocated to its home node, where other processes may subsequently access it by calling the *GlobalMap* function. The object will be destroyed when some process, host or PIM, calls *GlobalFree* on it, or the computation terminates.

For simplicity, our sharing model supports only a single copy of the data and will block to enforce serialized access if necessary. All access control is serialized through the home node. In such a basic environment, careless use of the *GlobalMap* function can result in deadlock; this is regarded as a programming error.

The distributed-shared-memory mechanism outlined above is intended for simple block-oriented data sharing, for applications where bandwidth is a more appropriate metric than latency. More flexible and finer-grained access is available via the parcel mechanism, which may be invoked either explicitly with user-mode access to the network interface, or implicitly, by the PIM kernel in response to an access fault. Note that our remote-access model permits access to portions of existing global segments which are not mapped to physical memory on the local PIM node, at higher cost.

## 8 Summary and Conclusion

This paper has described the memory management requirements for DIVA, a PIM-based architecture incorporating PIMs as the only memory for a conventional host processor. Two goals of the DIVA project impose fundamentally new requirements on memory management: DIVA PIMs must perform pointer accesses within memory, and they must support both smart-memory functionality as well as conventional memory accesses. The adoption of a globally shared address space for both host and PIM nodes allows free use of pointer-based data structures. Careful partitioning of complex memory-management tasks such as paging and swapping between the host and PIM node kernel allows a single host processor to supervise many PIM nodes without overload.

*Acknowledgments.* The authors wish to thank the DIVA research group at University of Notre Dame, headed by Peter Kogge and Jay Brockman, for their continuing contributions to the DIVA project. This research was supported by DARPA contract F30602-98-2-0180.

## References

1. D. Elliot, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing processors in memory. *IEEE Design and Test of Computers*, pages 32–41, January-March 1999.
2. M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *IEEE Computer*, pages 23–31, April 1995.
3. M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA: A data-intensive architecture. In *Proc. of SC '99*, November 1999.
4. P. Kogge. The EXECUBE approach to massively parallel processing. In *Int. Conf. on Parallel Processing*, August 1994.
5. Mitsubishi. M32R/D series: 32-bit RISC processor, on-chip DRAM. http://www.mitsubishi-chips.com/data/datasheets/mcus/m32rdgrp.html, May 1999.
6. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent DRAM: IRAM. *IEEE Micro*, April 1997.
7. A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proc. of the International Symposium on Computer Architecture*, May 1996.
8. A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *Proc. of the Symposium on High-Performance Computer Architecture*, May 1995.
9. T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

# Exploiting On-chip Memory Bandwidth in the VIRAM Compiler

David Judd, Katherine Yelick, Christoforos Kozyrakis, David Martin, and David Patterson

Computer Science Division
University of California [*]
Berkeley, CA 94720, USA
{dajudd,yelick,kozyraki,dmartin,pattrsn}@cs.berkeley.edu
http://iram.cs.berkeley.edu/

**Abstract.** Many architectural ideas that appear to be useful from a hardware standpoint fail to achieve wide acceptance due to lack of compiler support. In this paper we explore the design of the VIRAM architecture from the perspective of compiler writers, describing some of the code generation problems that arise in VIRAM and their solutions in the VIRAM compiler. VIRAM is a single chip system designed primarily for multimedia. It combines vector processing with mixed logic and DRAM to achieve high performance with relatively low energy, area, and design complexity. The paper focuses on two aspects of the VIRAM compiler and architecture. The first problem is to take advantage of the on-chip bandwidth for memory-intensive applications, including those with non-contiguous or unpredictable memory access patterns. The second problem is to support that kinds of narrow data types that arise in media processing, including processing of 8 and 16-bit data.

## 1 Introduction

Embedded processing in DRAM offers enormous potential for high memory bandwidth without high energy consumption by avoiding the memory bus bottlenecks of conventional multi-chip systems [FPC+97]. To exploit the memory bandwidth without expensive control and issue logic, the IRAM project at U.C. Berkeley is exploring the use of vector processing capabilities in a single-chip system called VIRAM, designed for multimedia applications [PAC+97]. Studies of hand-coded VIRAM benchmarks show that performance on a set of multimedia kernels exceeds that of high-end DSPs and microprocessors with media-extensions [Koz99,Tho00]. In this paper, we demonstrate that a vectorizing compiler is also capable of exploiting the vector instructions and memory bandwidth in VIRAM.

[*] This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056, by the California State MICRO Program, and by the Department of Energy. The VIRAM compiler was built using software provided by Cray, Inc.

The technology for automatic vectorization is well understood within the realm of scientific applications and supercomputers. The VIRAM project leverages that compiler technology with some key differences that stem from differences in the application space of interest and in the hardware design: the VIRAM processor is designed for multimedia benchmarks and with a primary focus on power and energy rather than high performance. There is some overlap between the algorithms used in multimedia and scientific computing, e.g., matrix multiplication and FFTs are important in both domains. However, multimedia applications tend to have shorter vectors, and a more limited dynamic range for the numerical values, which permits the use of single-precision floating-point as well as integer and fixed-point operations on narrow data types, such as 8, 16, or 32 bit values. VIRAM has features to support both narrow data types and short vectors.

## 2 Overview of the VIRAM Architecture

### 2.1 The Instruction Set

The VIRAM instruction set architecture (ISA) [Mar99] extends the MIPS ISA with vector instructions. It includes integer and floating-point arithmetic operations, as well as memory operations for sequential, strided, and indexed (scatter/gather) access patterns. The ISA specifies 32 vector registers, each containing multiple vector elements. Each vector instruction defines a set of operand vectors stored in the vector register file, a vector length, and an operation to be applied element-wise to the vector operands. Logically, the operation described by an instruction may be performed on all the vector elements in parallel. Therefore, we use the abstract notion of a *virtual processor* in which there is one simple processor per vector element that executes the operation specified by each vector instruction.

The maximum number of elements per vector register is determined by two factors: the total number of bits in a register and the width of the elements on which operations are being performed. For example, in the VIRAM processor a vector register holds 2K bits, which corresponds to 32 64-bit elements, 64 32-bit elements, or 128 16-bit elements. The VIRAM ISA supports arithmetic and memory operations on these three data widths. The bit width of the elements is known as the *virtual processor width* (VPW) and may be set by the application software and changed as different data types are used in the application.

Apart from narrow data types, multimedia applications frequently use fixed-point and saturated arithmetic. Fixed-point arithmetic allows decimal calculations within narrow integer formats, while saturation reduces the error introduced by overflow in signal processing algorithms. The VIRAM architecture supports both features with a set of vector fixed-point add, multiply, and fused multiply-add instructions. Programmable scaling of the multiplication result and four rounding modes are used to support arbitrary fixed-point number formats. The width of the input and output data for multiply-add are the same for these operations, hence all operands for this instruction can be stored in regular vector registers. There is no need for extended precision registers or accumulators, and this simplifies the use of these instructions. The maximum precision of calculations can be set by selecting the proper virtual processor width.

To enable efficient vectorization of conditional statements, the ISA includes a vector flag register file with 32 registers. Each register consists of a bit vector with one bit per vector element, which may be applied as a mask to the majority of vector operations. The same flag registers are used to support arithmetic exceptions, as well as software-controlled speculation of both load and arithmetic operations.

## 2.2 The VIRAM Processor

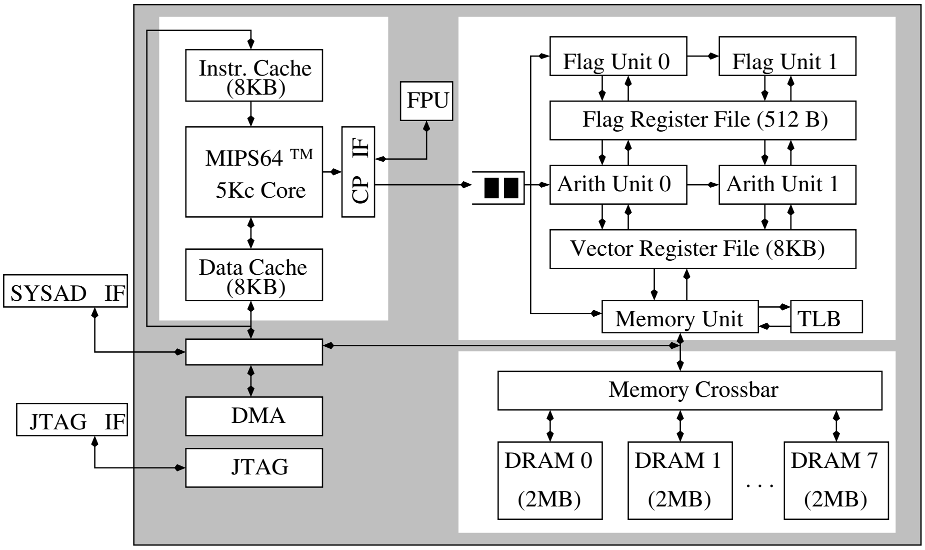


**Fig. 1.** The Block Diagram of the VIRAM Processor.

The VIRAM processor chip is an implementation of the VIRAM architecture designed at U.C. Berkeley [KGM$^+$00]. Its block diagram is presented in Figure 1. It includes a simple in-order (scalar) MIPS processor with first level caches and floating-point unit, a DMA engine for off-chip access, an embedded DRAM memory system, and a vector unit which is managed as a co-processor. Both the vector and scalar processors are designed to run at 200 MHz using a 1.2V power supply, with a power target of 2 Watts [Koz99].

The vector unit of the VIRAM processor includes one floating-point and two integer arithmetic units. Each arithmetic unit contains a 256-bit datapath, which can be used to execute 4 64-bit operations, 8 32-bit operations, or 16 16-bit operations simultaneously. Thus, the virtual processors that one may imagine acting on an entire vector register are implemented in practise by a smaller set of vector *lanes*, which execute a vector operation by computing a subset of the elements in parallel in each clock cycle. With 256 bits datapath width, the vector unit has 4 64-bit lanes, which can be viewed as 8 32-bit lanes or (for integer operations) 16 16-bit lanes.The ISA allows for 32 and 64-bit floating-point as well as 8, 16, 32, and 64-bit integer operations. The VIRAM processor supports only 32-bit floating-point and 16-, 32-, and 64-bit integer operations.[1] VIRAM's peak performance is 1.6 GFLOPS for 32-bit floating-point, 3.2 GOPS for 32-bit integer operations, and 6.4 GOPS for 16-bit integer operations.

The vector unit uses a simple, single-issue, in-order pipeline structure for predictable performance. The combination of pipelined instruction startup and chaining, the vector equivalent of operand forwarding, enables high performance even with short vectors. To avoid large load-use delays due to the latency of DRAM memory, the worst-case latency of a on-chip DRAM access is included in the pipeline and the execution of arithmetic operations is delayed by a few pipeline stages. This hides the latency of accessing DRAM memory for most common code cases.

There are 16 MBytes of on-chip DRAM in the VIRAM processor. They are organized in 8 independent banks, each with a 256-bit synchronous interface. A DRAM bank can service one sequential access every 6.5ns or one random access every 25ns. The on-chip memory is directly accessible from both the scalar and vector instructions using a crossbar interconnect structure with peak bandwidth of 12 GBytes/s. Instruction and data caches are used for scalar accesses, while references from the vector unit are served directly by the DRAM macros. The memory unit in the vector coprocessor can exchange 256 bits per cycle with the DRAM macros for sequential accesses, or up to four vector elements per cycle for strided and indexed accesses. It also includes a multi-ported TLB for virtual memory support.

## 3 Compiler Overview

The VIRAM compiler is based on the Cray vectorizing compiler, which has C, C++ and Fortran front-ends and is used on Cray's supercomputers. In addition to vectorization, the compiler performs standard optimizations including constant propagation and folding, common subexpression elimination, in-lining, and a large variety of loop transformations such as loop fusion and interchange. It also performs outer loop vectorization, which is especially useful for extracting large degrees of parallelism across images in certain multimedia applications. Pointer analysis in C and C++ are beyond the scope of this paper, so we use the Cray compiler strategy of requiring "restrict" pointers on array arguments to indicate they are unaliased.

[1] The fused multiply-add instructions in the VIRAM ISA are implemented in the VIRAM processor for fixed-point operations but not floating-point. These instructions are not targeted by the compiler and will therefore not be considered here.

The strategy in this project was to re-use as much compiler technology as possible, innovating only where necessary to support the VIRAM ISA. The Cray compiler has multiple machines targets, including vector machines like the Cray C90 and parallel machines like the T3E, but it did not support the MIPS architecture, so our first step was to build a MIPS back-end for scalar code and then a VIRAM code generator for vectorized code. Differences in VIRAM and Cray vector architectures lead to more interesting differences:

1. VIRAM supports narrow data types (8, 16, and 32 bits as well as 64 bits). Cray vector machines support 32 and 64 bit types, but, historically, the hardware ran 32-bit operations at the same speed as 64-bit, so there was no motivation to optimize for narrower types. In VIRAM using narrow data types leads to higher peak performance, as each 64-bit datapath can execute multiple narrower operations in parallel.
2. In both Cray and VIRAM architectures, conditional execution is supported by allowing a vector instruction to be computed with a set of flags (or masks) that effectively turns off the virtual processors at those positions. VIRAM treats flags as 1-bit vector registers, while the Cray machines treat them as 64 or 128-bit scalar values.
3. VIRAM has special support for fixed-point arithmetic, saturation and various rounding modes for integer operations, which are not provided on Cray machines.
4. VIRAM allows for speculative execution of vector arithmetic and load instructions, which is particularly important for vectorizing loops with conditions for breaking out of the loop, e.g., when a value has been found.

This list highlights the differences in the architectures as they affect compilation. In this paper we focus mainly on the first of these issues, i.e., generation of code for narrow data widths, which is critical to the use of vector processing for media processing. The last two features, fixed-point computations and speculative execution, have no special support in our compiler, and while the flag model resulted in somewhat simpler code generation for VIRAM, the difference is not fundamental. In VIRAM the vector length register implicitly controls the set of active flag values, whereas the Cray compiler must correctly handle trailing flag bits that are beyond the current vector length.

## 4 On-chip Memory Bandwidth

The conventional wisdom is that vector processing is only appropriate for scientific computing applications on high-end machines that are expensive in part because of their SRAM-based, high bandwidth, memory systems. By placing DRAM and logic on a single chip, VIRAM demonstrates that vector processing can also be used for application domains that demand lower cost hardware. In this section, we use a set of different implementations of the dense matrix-vector multiplication (MVM) benchmark to explore the question of how well VIRAM supports memory-intensive applications. Although MVM is a simple compilation problem, it does not perform well on conventional microprocessors due to

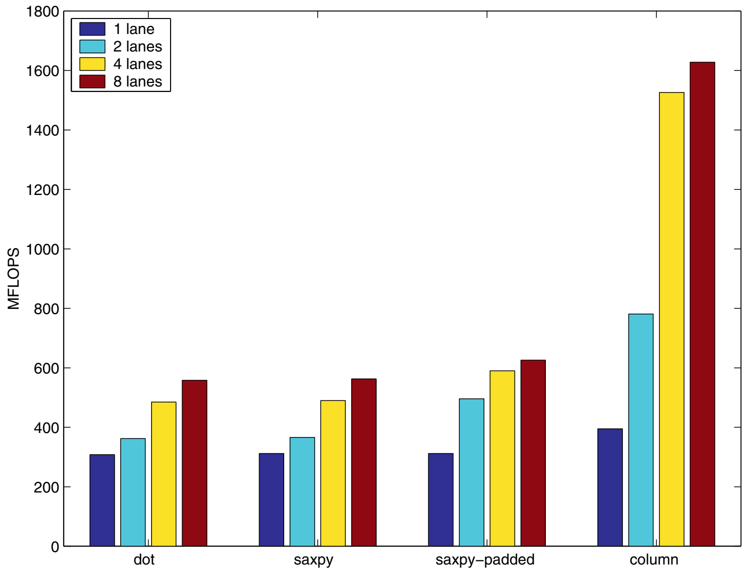


**Fig. 2.** Performance of $64 \times 64$ Single-precision Floating-point Matrix-Vector Multiplication on the VIRAM Processor.

the high memory bandwidth requirements. There are only two floating-point operations per matrix element.

When the matrix uses a row-major layout, matrix vector multiplication can either be organized as a set of dot products on the rows of the matrix or a set of saxpy's on the columns of the matrix. The dot products have an advantage in using unit stride access (sequential) to the matrix, while saxpy's require strided accesses. Strided and indexed vector accesses are slower than unit stride, because they can only fetch four elements per cycle, regardless of the virtual processing width. A unit stride load can always fetch 256 bits per cycle, which is eight 32-bit elements for example. In addition, strided and indexed accesses may cause DRAM bank conflicts which stall the vector unit pipeline. On the other hand, the dot product involves a reduction that is less efficient, because it involves operations on short vectors near the end of the reduction. For very long vectors, the final stages of the reduction are amortized over more efficient full vector operations, so the impact on efficiency is negligible.

Figure 2 shows the performance in MFLOPS of single-precision floating-point matrix vector multiplication routine on a $64 \times 64$ matrix. The numbers are taken for the VIRAM processor implementation by varying the number of 64-bit lanes from 1 to 8. Our performance results are produced using a simulator that gives a cycle-accurate model of the VIRAM vector unit. The first of these groups is a dot-product implementation using unit stride. Not surprisingly, with only 64 elements per vector, efficiency is somewhat low due the percentage of time spent on reductions of short vectors. The second version shows a saxpy implementation, where the accesses to the matrix are the stride of the matrix dimension (64). If one pads the array with an extra element, giving it a prime stride of 65, that reduces memory bank conflicts, resulting in the third group of numbers. Finally, if we use a column-major layout for the matrix, a saxpy implementation can be used with unit stride, which produces the best performance. We note that while is may be attractive to consider only using the column-based layout, another common kernels used in multimedia application is vector-matrix multiplication, which has exactly the opposite design constraints, so a row layout is preferred there.

As indicated by this data, an 8-lane vector unit does not perform well with only 8 banks. Even with 4 lanes there are significant advantages to having more banks for the saxpy implementations with the row-major matrix layout. More DRAM banks reduce the likelihood of bank conflicts for strided memory accesses. A similar effect can be achieved by organizing the existing banks in a hierarchical fashion with sub-banks, which would allow overlapping of random accesses to a single bank [$Y^+97$]. Unfortunately, hierarchical DRAM bank technology was not available to us for the VIRAM processor, but we expect it to be available in the next generation of embedded DRAM technology.

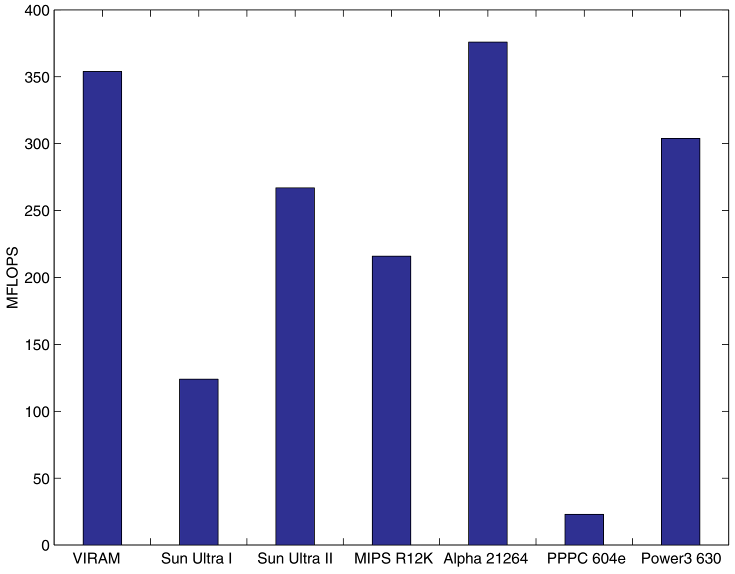


**Fig. 3.** Performance of 100 × 100 Double-precision Floating-point Matrix-Vector Multiplication.

Nevertheless, the performance is impressive even for the 4-lane, 8 bank VIRAM design when compared to conventional microprocessors with higher clock rates, higher energy requirements, more total area per microprocessor, and many more chips in the system for caches and main memory. Figure 3 presents the performance of the hand-coded vendor implementation of $100 \times 100$ double-precision MVM as reported in the LAPACK manual for a set of high performance server and workstation processors [ABB+99]. A column-major layout is used with all of them. The VIRAM processor does not support double-precision operations on its floating-point datapaths because they are of no practical use for multimedia applications. Still, we were able to calculate the performance of a four lane VIRAM processor with support for double-precision using a simulator.

From the six other processors, only the hand-coded Alpha 21264 outperforms compiled code on VIRAM by merely 6%. The rest of the processors perform 1.3 to 15 times worse, despite their sophisticated pipelines and SRAM based cache memory systems. Note that for larger matrices VIRAM outperforms all these processors by larger factors. The performance of server and workstation processors is reduced for larger matrices as they no longer fit in their first level caches. On the other hand, the vector unit of VIRAM accesses DRAM directly and incurs no slow-down due to caching effects, regardless of the size of the matrix. In addition, multiplication of larger matrices leads to operations on longer vectors, which amortize better the performance cost of the final stages for reduction operations in the saxpy implementation. Hence, the performance of the VIRAM processor actually increases with the size of the matrix.

## 5 Narrow Data Types

One of the novel aspects of the VIRAM ISA is the notion of variable width data which the compiler controls by setting the virtual processor width (VPW). The compiler analyzes each vectorizable loop nest to determine the widest data width needed by vector operations in the loop and then sets the VPW to that width. The compiler uses static type information from the variable declaration rather than a more aggressive, but also more fragile, variable width analysis [SBA00].

The process of computing the VPW is done with multiple passes over a loop nest. On the first pass, the VPW is assumed to be 64 bits, and the loop nest is analyzed to determine whether it is vectorizable. At the same time, the widest data type used within the loop is tracked. If the maximum data width is determined to be narrower than 64 bits, vectorization is re-run on the loop nest with the given estimate of VPW (32 or 16 bits). The reason for the second pass is that narrowing the data width increases the maximum available vector length, and therefore allows more loops to be executed as a single sequence of vector instructions without the strip-mining loop overhead.

The above strategy works well for 32-bit data types (either integer or floating-point) but does not work well for 16 bits in standard C. The reason is that the C language semantics require that, even if variables are declared as 8-bit characters or 16-bit shorts, most operations must be performed as if they were in 32 bits. To enforce these semantics, the compiler front-end introduces type casts rather aggressively, making it difficult to recover information about narrower types in code generation. Our initial experiments found that only the most trivial of loops, such as an array initialize, would run at a width narrower than 32 bits. As a result, we introduced a compiler flag to force the width to 16 bits; the compiler will detect certain obvious errors such as the use of floating-point in 16 bit mode, but trusts the programmer for the bit-width of integer operations.

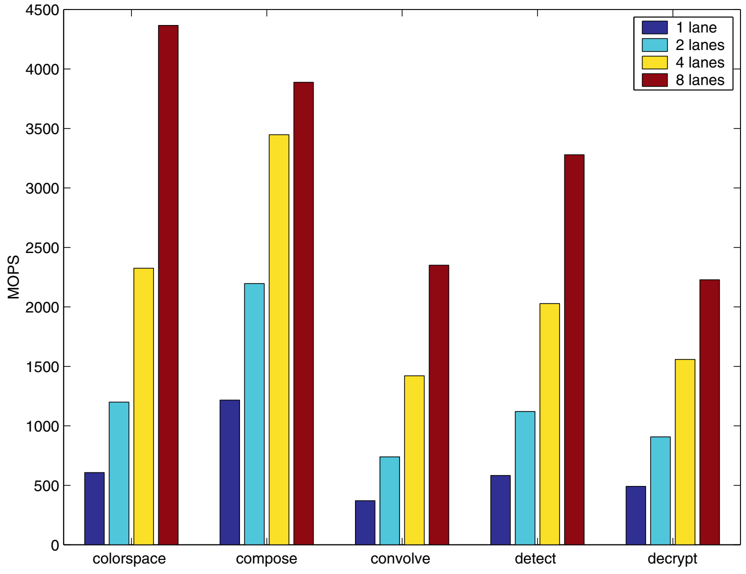


**Fig. 4.** Performance of Integer Media Kernels on VIRAM.

Figures 4 and  5 shows the performance of several media processing kernels on a VIRAM configuration with 16 memory banks and varying numbers of 64-bit lanes from 1 to 8. Figure 4 presents integer benchmarks taken from the UTDSP benchmark suite [LS98], while Figure 5 includes single-precision floating-point kernels.

- `colors` (colorspace conversion) is a simple image processing kernels that converts from the RGB to the YUV format. It reads and writes 8-bit data and operates internally on it as 16-bit data. Strided memory operations are convenient for selecting pixels of the same color that are interleaved in RGB format in the image. Loads of 8-bit data while the VPW is set to 16 bits, fetches each 8-bit value into a single 16-bit vector element, without any shuffling or expanding of data needed.
- `compose` (image composition) is similar to `colors` in the data types, but can be performed using unit stride memory accesses only.
- `convlv` is an image convolution, which like `colorspace` performs a 16-bit integer computation on images stored in RGB format using strided memory accesses. Each output pixel is computed by multiplying and summing the

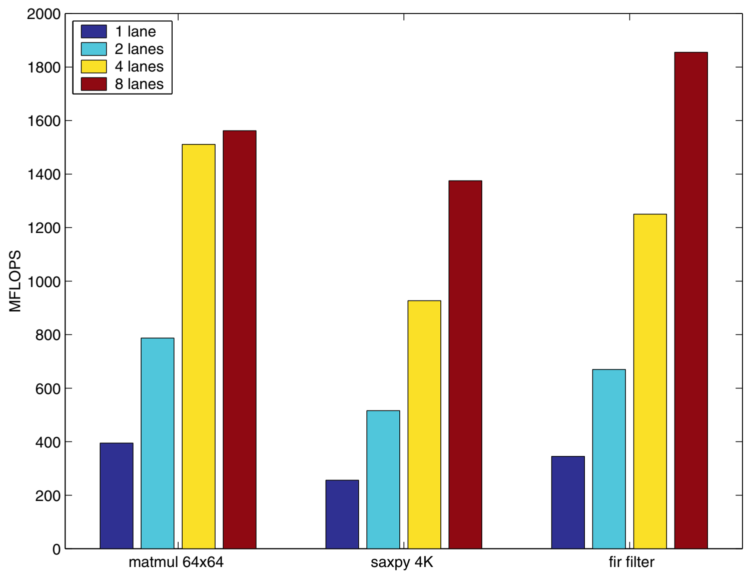


**Fig. 5.** Performance of Single-Precision Floating-point Media Kernels on VIRAM.

pixels in a 3x3 region. The source code contains 4 nested loops, and it is critical that the compiler vectorize an outer loop to obtain reasonable vector lengths. By default, the compiler merges the middle two loops and vectorizes them while unrolling the innermost one. The performance shown here improves on that default strategy by unrolling the two inner loops in source code, while the compiler vectorizes the 2nd loop that goes over rows of the image.

– `detect` is an edge detection algorithm for images and `decrypt` performs IDEA decryption. Both of these use a mixture of 16-bit and 32-bit integer operations.
– `FIR` is an FIR filter, `saxpy1` and `saxpy2` are, respectively, 64 element and 1024 element saxpy's. `Matmul` is a 64x64 matrix multiplication. All of these use 32-bit floating-point computations.

Although hand-optimized VIRAM code is not available for all of these kernels in this form, two floating-point kernels are available and are quite close in performance. The hand-coded performance numbers for a 4-lane configuration with 8 memory banks are: 720 MFLOPS for `saxpy2` and 1,580 MFLOPS for `matmul`. Note that the performance in both cases depends on the size of the input data. The two versions of saxpy are included to show how performance improves with longer application level arrays, because they better amortize the time it takes to fill and drain the long vector pipeline at the beginning and at the end of the kernel. This shows again an advantage VIRAM for computations that operate on large images or matrices. VIRAM does not depend on caches, hence it performance often increases rather than decreases with increased data size.

The performance of kernels with 16-bit operations is limited primarily by the number of elements that can be fetched per cycle for non-unit stride accesses. The VIRAM processor fetches one element per lane for indexed and strided accesses, while a 64-bit lane can execute 4 16-bit arithmetic operation simultaneously. This mismatch exists due to the high area, power, and complexity cost of generating, translating, and resolving for bank conflicts more than one address per lane per cycle for strided and indexed accesses. Unit stride accesses, due to their sequential nature, can fetch multiple elements with a single address for all lanes. Bank conflicts are also responsible for some performance degradation.

## 6 Related Work

The most closely related compiler effort to ours is the vectorizing compiler project at the University of Toronto, although it does not target a mixed logic and DRAM design [LS98]. Compilers for the SIMD microprocessor extensions are also related. For example, Larsen and Amarsinghe present a compiler that uses *Superword Level Parallelism* (SLP) for these machines; SLP is identical to vectorization for many loops, although it may also discover vectorization across statements, which typically occurs if a loop has been manually unrolled.

Several aspects of the VIRAM ISA design make code generation simpler than with the SIMD extensions [PW96,Phi98]. First, the VIRAM instructions are independent of the datapath width, since the compiler generates instructions for the full vector length, and the hardware is responsible for breaking this into datapath size chunks. This simplifies the compilation model and avoids the need to recompile if the number of lanes changes between generations. Indeed, all of the performance numbers below use a single executable when varying the number of lanes. Second, VIRAM has powerful addressing modes such as strided and indexed loads and stores that eliminate the need for packing and unpacking. For example, if an image is stored using 8-bits per pixel per color, and the colors are interleaved in the image, then a simple array expression like `a[3*i+j]` in the source code will result in a strided vector load instruction from the compiler.

## 7 Conclusions

This paper demonstrates that the performance advantages of VIRAM with vector processing do not require hand-optimized code, but are obtainable by extending the vector compilation model to multiple data widths. Although some programmer-control over narrow data types was required, the programming model is still easy to understand and use. VIRAM performs well and demonstrates scaling across varying numbers of lanes, which is useful for obtaining designs with lower power and area needs or for high performance designs appropriate for a future generations of chip technology. The compiler demonstrates good performance overall, and is often competitive with hand-coded benchmarks for floating-pointer kernels. The availability of high memory bandwidth on-chip makes a 2 Watt VIRAM chip competitive with modern microprocessors for bandwidth-intensive computations. Moreover, the VIRAM instruction set offers an elegant compilation target, while the VIRAM implementation allows for scaling of computation and memory bandwidth across generations through the addition of vector lanes and memory banks.

**Acknowledgments**

The authors would like to thank the other members of the VIRAM group and Corinna Lee's group at the University of Toronto who provided the UTDSP benchmarks. We are also very grateful for the support provided by the Cray, Inc. compiler group in helping us use and modify their compiler. IBM Microelectronics provided the manufacturing technology for the VIRAM processor, while MIPS Technologies designed the MIPS processor it includes. Avanti Corp. provided the CAD tools used for the design.

## References

ABB+99. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. *LAPACK Users' Guide: Third Edition.* SIAM, 1999.

FPC+97. R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *the 24th Annual International Symposium on Computer Architecture*, pages 327–337, Denver, CO, June 1997.

KGM+00. C.E. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and D. Yelick. VIRAM: A Media-oriented Vector Processor with Embedded DRAM. In *the Conference Record of the 12th Hot Chips Symposium*, Palo Alto, CA, August 2000.

Koz99. Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report UCB//CSD-99-1059, University of California, Berkeley, July 1999.

LS98. C.G. Lee and M.G. Stoodley. Simple vector microprocessors for multimedia applications. In *31st Annual International Symposium on Microarchitecture*, December 1998.

Mar99. D. Martin. *Vector Extensions to the MIPS-IV Instruction Set Architecture.* Computer Science Division, University of California at Berkeley, January 1999.

PAC+97. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for Intelligent DRAM: IRAM. *IEEE Micro*, 17(2):34–44, April 1997.

Phi98. M. Phillip. A second generation SIMD microprocessor architecture. In *Proceedings of the Hot Chips X Symposium*, August 1998.

PW96. A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

SBA00. M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.

Tho00. Randi Thomas. An architectural performance study of the fast fourier transform on VIRAM. Technical Report UCB//CSD-99-1106, University of California, Berkeley, June 2000.

$Y^+97$. T. Yamauchi et al. The hierarchical multi-bank DRAM: a high-performance architecture for memory integrated with processors. In *the Proceedings of the 17th Conf. on Advanced Research in VLSI*, Ann Arbor, MI, Sep 1997.

# FlexCache: A Framework for Flexible Compiler Generated Data Caching

Csaba Andras Moritz[1], Matthew I. Frank[2], and Saman Amarasinghe[2]

[1] University of Massachusetts,
Electrical and Computer Engineering,
Amherst, Ma 01002
andras@ecs.umass.edu

[2] Massachusetts Institute of Technology,
Laboratory for Computer Science,
Cambridge, Ma 02139
{mfrank,saman}@lcs.mit.edu

**Abstract**
This paper describes our work in progress on FlexCache, a framework for flexible, compiler generated data caching. FlexCache substitutes the tag-memory and cache controller hardware with a compiler managed tag-like data structure, address translation, and tag-check. This allows the division of the data-array into separately controlled partitions, allows the selection of cache line sizes, the support of highly associative mappings, and the selection of various replacement policies on a per program basis.

FlexCache leverages compile-time static information to selectively virtualize memory, to eliminate cache-tag accesses, and to guide the replacement of conflicting cache lines. For the applications studied the FlexCache compiler techniques eliminate in average more than 90% of the cache-tag lookups, enabling the support of highly associative caching schemes. Even without any hardware support, FlexCache can outperform fixed hardware caches by improving caching effectiveness, eliminating mapping conflicts, and eliminating the cache pollution caused by register spills. The beauty of FlexCache is that its core techniques could be augmented with additional software techniques and/or hardware support (i.e., special instructions) to look into optimizing data caching in areas such as low-power, real-time systems, and high-performance microprocessors.

## 1 Introduction

Several recent project proposals focus on issues related to memory resources, as memory resources still represent the most important bottleneck in computing devices, in respect to power-consumption, performance, and area occupied [4]. Because of rapidly changing requirements for optimal designs, these projects emphasize on memory systems that are reconfigurable or optimizable to some extent. Good example is Smart Memories [6] that successfully maps architectures with different memory requirements to a low-level computing fabric. The

Berkeley IRAM project [9] improves memory latency by designing a processor in DRAM process and includes the DRAM memory on-chip. The MIT Raw project [10] focuses instead on compiler technology to control low-level hardware resources to optimize the performance of memory accesses. The FlexRAM project [5] moves computation closer to the DRAM in an attempt to reduce memory latency for memory bound applications by combining processing and DRAM on the same chip. The Active Pages system [7] shifts data-intensive computations to the memory system by combining FPGAs with DRAM. A new hardware cache is shown in [8] that divides the SRAM storage into several partitions and use it as prefetch buffer, compiler controlled memory, or lookup buffer. This design is shown to benefit media applications with up to 20% performance improvement. Digital Signal Processors such as the Texas Instruments TMS320C62XX added hardware support to use SRAM data storage as both cache and main memory. To cope with the long DRAM latencies in future microprocessors a fully associative secondary cache with software replacement/control is suggested in [3]. A software replacement algorithm is shown to achieve miss-rate reductions from 8% to 85% compared to a 4-way LRU.

This paper describes FlexCache, a framework for flexible, compiler generated data caching. FlexCache takes the flexibility of the above mentioned memory systems one step further. Instead of building upon conventional caching architectures, FlexCache builds upon a flexible software platform with the possibility of adding hardware resources as needed. Hardware support can be added in form of new instructions to reduce the overhead of the software schemes, but without altering the flexibility and programmability provided.

FlexCache substitutes the tag-memory and the cache controller with a compiler managed tag-like data structure, address translation, and tag-checks. It allows the division of the data-array into separately controlled partitions, the selection of cache line sizes, the support of highly associative mappings, and the selection of various replacement policies. Because FlexCache leverages static information available during compilation, it can improve caching efficiency compared to conventional caches. Eliminating the hardware for managing cache-tags and providing direct access to fast SRAM can also simplify the pipeline and reduce design overhead. FlexCache incorporates new compiler techniques called Hot Pages, succesfully eliminating tag-checks, making possible the design of highly associative caches. The FlexCache approach is different from many other compiler systems that aimed compiler optimizations at reducing the impact of memory latency in the context of conventional hardware caches.

FlexCache uses four main techniques in its base implementation to reduce overheads during the hit case. First, it can avoid caching when not necessary by mapping (at compile time) some memory accesses directly into fast SRAM memory. Example of such accesses are register spills that otherwise pollute caches and can cause significant performance degradation [2]. Second, it uses for the first time compiler techniques enabled by global pointer analysis to identify and speculatively reuse previously calculated cache mappings at runtime. In our software implementation this optimization reduces the software overhead for the hit

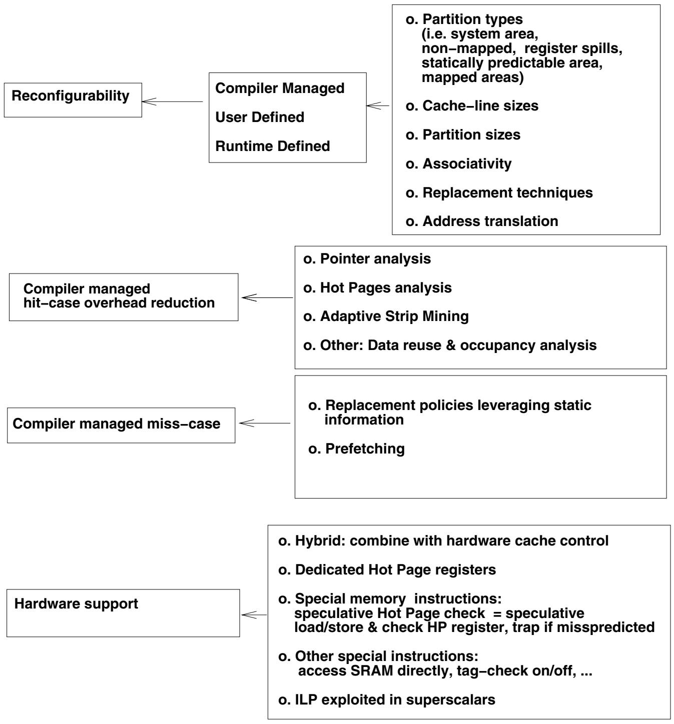

Reconfigurability
Compiler Managed
User Defined
Runtime Defined
o. Partition types (i.e. system area, non−mapped, register spills, statically predictable area, mapped areas)
o. Cache−line sizes
o. Partition sizes
o. Associativity
o. Replacement techniques
o. Address translation
Compiler managed hit−case overhead reduction
o. Pointer analysis
o. Hot Pages analysis
o. Adaptive Strip Mining
o. Other: Data reuse & occupancy analysis
Compiler managed miss−case
o. Replacement policies leveraging static information
o. Prefetching
Hardware support
o. Hybrid: combine with hardware cache control
o. Dedicated Hot Page registers
o. Special memory instructions: speculative Hot Page check = speculative load/store & check HP register, trap if misspredicted
o. Other special instructions: access SRAM directly, tag−check on/off, ...
o. ILP exploited in superscalars

**Fig. 1.** Overview of the FlexCache framework. FlexCache can combine compiler analysis, user input, or runtime profiling for cache configuration selection. New compiler techniques are proposed to reduce software overheads in software managed caching, to manage the miss-case efficiently, and to eliminate tag-checks. Hardware support can be added in form of new speculative memory instructions, direct access to the cache data-array, and in form of dedicated Hot Page registers. Additionally, a FlexCache proposes to incorporate hardware cache-control providing the ability to handle some memory accesses in conventional manner.

case to four instructions per load (in a pure software implementation running on a single-issue processor), independent of the sophistication of the underlying caching strategy. The four instruction overhead can be reduced to two instructions on multiple issue processors, and can be completely eliminated with special Instruction Set Architecture (ISA) extensions. Third, for dense array accesses the system uses *adaptive strip mining* to reduce the number of tag checks from once per memory access to once per cache line. Finally, the FlexCache system allows the user to select cache-line size, associativity and replacement policy for each program, reducing the miss rate.

Additional compiler techniques could be developed to automate the selection of these parameters. This paper mainly focuses on identifying the critical components of the FlexCache system and leaves out the compiler techniques required to automate the configuration selection process.

An overview of the FlexCache framework is presented in Figure 1. A FlexCache based cache can combine compiler analysis, user input, or runtime profiling for optimal cache configuration selection. FlexCache incorporates new compiler techniques to reduce software overheads in software managed caching, to manage the miss-case efficiently (i.e., by leveraging compiler knowledge we can design application specific replacement policies), and to eliminate cache tag-checks. Hardware support can be added in form of new speculative memory instructions, direct access to the cache data-array, support for access to different type of memory partitions, and in form of dedicated Hot Page registers. Additionally, FlexCache proposes to incorporate hardware cache-control, in a hybrid cache solution, providing the ability to handle some memory accesses in conventional way (i.e., same manner as in a hardware cache).

Our preliminary results demonstrate that compile time analysis can eliminate a large portion (90% in average for the applications studied) of the cache-tag lookups. Even without any hardware support, FlexCache can outperform fixed hardware caches by improving caching effectiveness, eliminating mapping conflicts, and eliminating the cache pollution caused by register spills. Because all the main FlexCache features are software based, a FlexCache solution is application specific and can be easily retargeted to focus on different design aspects such as low power consumption, high performance, or high predictability of memory accesses. FlexCache can also be the right solution to add data caching for FPGA based systems if it is incorporated in a silicon compilation system such as described in [1].

The remainder of this paper is organized as follows. Section 2 and 3 describes the components of the FlexCache system. Section 4 presents suggestions for hardware support. Section 5 gives our preliminary experimental results. Section 6 concludes the paper.

## 2 FlexCache Runtime System

This paper assumes an architecture with a local SRAM memory and an external large DRAM. Address translation is the mapping of program addresses into SRAM addresses. This mapping can be implemented by using a table lookup similar to the page table lookup used in virtual memory systems. The mapping is done at a *line* granularity. A line is a contiguous address range in both SRAM and program memory. Each translation table entry contains both a tag and a translation to a physical SRAM address.

The translation overhead if done in software varies for different table organizations, but requires at minimum, (1) calculating the line number from the program address, (2) calculating an address into the translation table, based on the line number, (3) loading the tag, (4) comparing the line number with the tag, (5) calculating the offset within the line, (6) loading the translation, (7) calculating the actual physical SRAM address from the translation and offset, (8) actually loading the data from SRAM. Associative caching schemes may require multiple table lookups and tag compares (steps 2-4). Hardware cache management schemes provide special hardware to perform all of the work for steps 1-7, usually in a single cycle. Section 3 describes our compiler optimizations to reduce the software overhead. The Hot Pages optimization eliminates steps 2, 3 and 6. ISA extensions implementing the remaining operations could eliminate all the remaining overhead, while keeping (i.e., not altering) the flexibility of the compiler based solution.

When the FlexCache system detects a cache miss (the software tag-check fails) it invokes a miss handler. Table entries that correspond to Hot Pages (cache-lines identified as hot during compile-time) are non-replaceable and are ignored during replacement. In the direct mapped cache there is only one line that can be replaced. If we map hot pages through this table we have no other option but to replace them in case of a conflict situation. However, we can rely on pointer analysis to guarantee that non hot page accesses will not access lines that currently map to hot pages and therefore we can move the mapping of hot pages outside the main mapping table. This improves the hit-case as cache lines predicted statically to be hot can only be replaced with other (static) hot lines.

As the gap between processing speeds and external DRAM latencies is rapidly widening, with DRAM latencies reaching thousands of processing cycles, managing cache misses in software becomes a feasible approach.

## 3 FlexCache Compiler

Traditionally, the compiler generates code assuming infinite memory. The compiler component of the FlexCache system removes this assumption by implementing caching in software. Figure 2 shows the flow of the augmented FlexCache compiler which includes the new phases required for caching.

This section shortly describes the compiler components of the FlexCache system. First, it describes pointer analysis, an analysis technique used to determine the location set list of each memory reference. Next, it describes the Hot Page analysis phase which divides memory references into groups called *hot page sets*. All references inside a hot page set will use the same register allocated translation, called a *hot page*, an important overhead reduction optimization. Figure 3 introduces an example which illustrates the steps the compiler performs for software managed caching.

**Pointer Analysis**

Pointer analysis is a compiler analysis which finds a conservative estimation for the set of data objects that a memory reference can refer to. The analysis is conservative in that some objects in the set may not be referenced. One standard application of pointer analysis is to determine dependence between memory references. In our FlexCache system, the analysis is used to guide the placement of data and for the hot page optimization. There are many variants of pointer analysis algorithms, that mainly differ in the precision at which memory references are disambiguated. The FlexCache system is based on a most precise type of

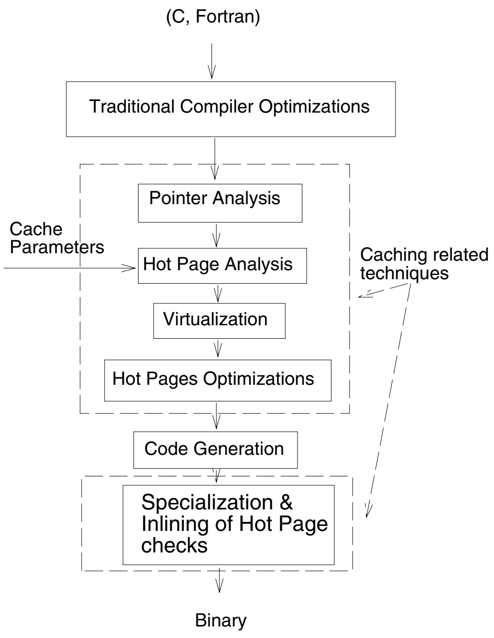


**Fig. 2.** Structure of the FlexCache Compiler

pointer analysis that is both flow-sensitive (i.e., takes control-flow into account) and context-sensitive (i.e., location sets may differ based on calling context).

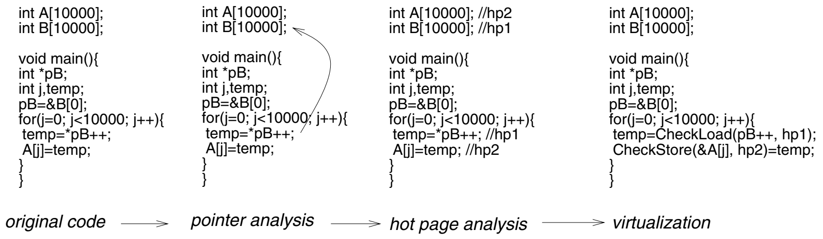
However, the FlexCache approach do not require this type of precision for correctness of execution, less precise but faster pointer analysis techniques could have been used, perhaps with somewhat less static prediction accuracy for cache accesses. Details about the pointer analysis used can be found in [11, 12].

**Hot Page Analysis**

The Hot Page analysis pass leverages the information provided by the pointer analysis pass to determine the location set of all memory accesses. The objective of the analysis is to identify if a program memory reference can reuse a previously translated virtual page description.

Our technique leverages static information about the locality of accesses, to implement a fast address translation. The compiler groups memory accesses into groups called *hot page sets*. Hot page sets are determined based on their location sets (information given by pointer analysis) in memory. Each hot page set contains references that can likely reuse the address translation saved for a specific memory line called a *hot page*.

The compiler algorithm has two phases. First, it finds data objects such as arrays and structures and maps these objects to hot page sets. Then, it traverses the control-flow graph of the program and maps memory accesses to existing hot page sets based on their location sets. For example, if the compiler determines



```
int A[10000];
int B[10000];

void main(){
int *pB;
int j,temp;
pB=&B[0];
for(j=0; j<10000; j++){
 temp=*pB++;
 A[j]=temp;
}
}
```

```
int A[10000];
int B[10000];

void main(){
int *pB;
int j,temp;
pB=&B[0];
for(j=0; j<10000; j++){
 temp=*pB++;
 A[j]=temp;
}
}
```

```
int A[10000]; //hp2
int B[10000]; //hp1

void main(){
int *pB;
int j,temp;
pB=&B[0];
for(j=0; j<10000; j++){
 temp=*pB++; //hp1
 A[j]=temp; //hp2
}
}
```

```
int A[10000];
int B[10000];

void main(){
int *pB;
int j,temp;
pB=&B[0];
for(j=0; j<10000; j++){
 temp=CheckLoad(pB++, hp1);
 CheckStore(&A[j], hp2)=temp;
}
}
```

**Fig. 3.** An example of how FlexCache implements software caching in the compiler: A.) Pointer analysis is used to determine the location sets of memory references, for example *pB* has the same set as *B*, B.) the *Hot Pages* analysis annotates memory references into hot page sets *hp1, hp2*, C.) the *virtualization* pass changes the memory references with procedure calls. The address translation *specialization* pass in the compiler back-end inlines *specialized* code for *CheckLoad, CheckStore*. This specialization is controlled by the hot page set annotations (i.e., *hp1, hp2*).

from the location set information that a load is accessing a location from a memory area allocated to an array, then it can (likely) reuse the address translation saved for the hot page set assigned to that array. Note that several location sets can be *hot* at the same time.

Each Hot Page has two pieces of information assigned to it: a program line number (the tag) and a physical frame, both saved into registers. A successful translation done through a Hot Page register will add only 4 cycles of overhead in a full software implementation: one cycle to extract the line number from the address, one cycle to compare to the tag stored in the `hp_vpn` register, one cycle to extract the line offset from the program address, and a final cycle to add the offset to the translated SRAM address held in the `hp_phys_frame` register. The next section will show additional techniques based on architectural extensions that eliminate the 4 cycles overhead.

If the Hot Page tag check fails, then a procedure call is made to the slower software runtime tag check routine, which takes about 23 cycles in our system. Alternatively, a hybrid system would rely on the hardware cache check in case of misspredicted Hot Page access.

As mentioned earlier, the Hot Page check itself could be speed up by implementing the hot page check in the instruction set of the processor.

The compiler determines which accesses should be cached in the virtualization phase. If an access is virtualized then it is substituted with a procedure call. The compiler can decide not to virtualize an access and map it directly to an unmapped portion of SRAM. These accesses are local and only cost 1 cycle (*i.e.,* as fast as having a hardware cache). The compiler maps scalar register spills to this area. Larger stack mapped objects are handled through the software caching system to avoid overloading the unmapped portion of SRAM.

The address translation for Hot Page references is customized at compile-time for the specific Hot Page page description (translation). No extra table lookup

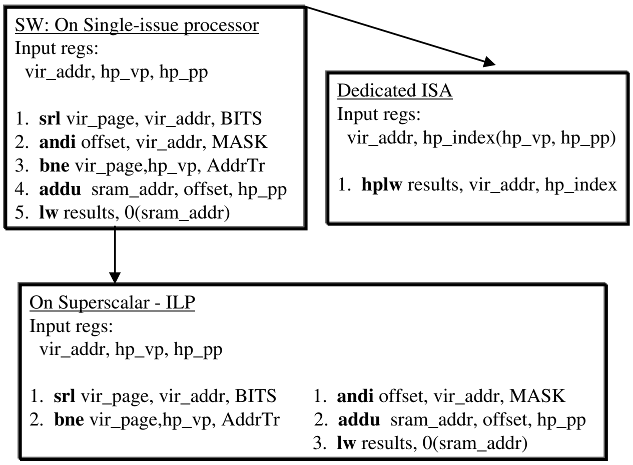

SW: On Single-issue processor
Input regs:
vir_addr, hp_vp, hp_pp
1. srl vir_page, vir_addr, BITS
2. andi offset, vir_addr, MASK
3. bne vir_page,hp_vp, AddrTr
4. addu sram_addr, offset, hp_pp
5. lw results, 0(sram_addr)
Dedicated ISA
Input regs:
vir_addr, hp_index(hp_vp, hp_pp)
1. hplw results, vir_addr, hp_index
On Superscalar - ILP
Input regs:
vir_addr, hp_vp, hp_pp
1. srl vir_page, vir_addr, BITS
2. bne vir_page,hp_vp, AddrTr
1. andi offset, vir_addr, MASK
2. addu sram_addr, offset, hp_pp
3. lw results, 0(sram_addr)

**Fig. 4.** Three different possible implementations of Hot Pages loads using a MIPS ISA. First, we have a load with 4 additional instructions overhead on single-issue processors. On a superscalar processor this overhead is reduced to two instruction per load because of the ILP in the check code. With the ISA extension the overhead is completely eliminated.

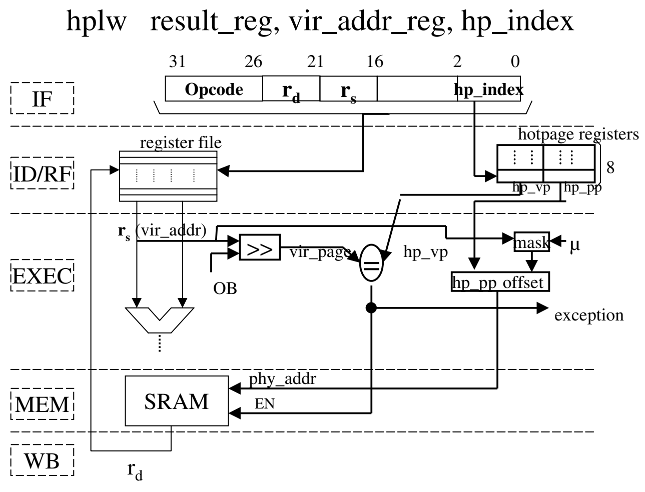
briefly described above, maintains the added flexibility and improved efficiency provided by the software techniques, in addition to a hit-case latency similar to that of conventional hardware based implementations. In a hybrid solution we can use hardware cache partitions for memory accesses that are not predictable or analyzable at compile-time (or memory accesses that are infrequent and hard to analyze). Example of such accesses are non-affine array accesses and references to complex data structures using pointers. More predictable references are dealt with using compiler managed memory areas or software caching, with added architectural support.

## 5 Experiments

We implemented the FlexCache system on a single-issue 5 stage pipeline microprocessor similar to MIPS R4000. A two level memory hierarchy based on a local SRAM and an off-chip DRAM is simulated. A local on-chip SRAM memory is attached to the processor. A large external RDRAM memory is necessary to solve large problems that exceed the size of the on-chip memory. The experiments are performed on a cycle-level simulator.

We have found that the FlexCache system has an average dynamic memory access prediction rate of 90% for the applications studied (see Table 1), eliminating the need of cache-tag lookups (or address translations) in most cases.

The FlexCache system is fully customizable that can benefit many applications. Applications have very different requirements that often cannot be ex-



• OB = # of bits in offset of hotpage vir_addr = $\log_2$(cache line size)
• μ = 000…..1111 , where # of 1's = OB
• hp_pp is the hotpage register containing the SRAM frame
• hp_vp is the hotpage register containing the virtual page

**Fig. 5.** Data-path of the *hplw* instruction. Note that the speculative addressing of the cache can be done with minimal hardware support. The HotPages registers containing the HotPages translations are used to reduce register pressure. A missprediction is handled as an exception and it is detected early in the Execution stage of the processor pipeline.

ploited in a fixed hardware caching system. In Figure 7 we show two applications with very different cache-line requirements. Execution times vary as much as 70%.

In Figure 6 we show that the software column associative scheme can eliminate many conflict misses that are produced in the direct mapped case. Compared with a fixed direct mapped hardware caching scheme, FlexCache can provide more associativity (i.e., because it is software managed) when needed. The Convolution program is three times faster with the software FlexCache system compared to a system using a fixed direct mapped hardware cache.

## 6 Conclusions and Future Work

This paper presented FlexCache, a framework for flexible, compiler managed data caching.

For the applications studied the FlexCache techniques eliminate in average more than 90% of the cache-tag lookups, enabling the support of highly associative caching schemes while providing full flexibility. We have shown that even

| Application | Total Cache Access. | Mispredicted | % optimized |
|---|---|---|---|
| Jacobi | 19392 | 505 | 97.3 |
| MxM | 12928 | 73 | 99.4 |
| Cholsky | 25440 | 4502 | 82.3 |
| Life | 2516 | 54 | 97.8 |
| Moldyn | 243740 | 11487 | 95.2 |
| Adpcm | 51202 | 420 | 99.1 |
| Sor | 14131 | 1024 | 92.7 |
| Vpenta | 6868 | 2321 | 66.2 |
| MedianF | 410018 | 21416 | 94.7 |

**Table 1.** Percentage of tag-checks (translation table lookups) optimized with the Hot Pages technique in a FlexCache system. The total percentage of tag checks eliminated is actually higher because a large portion of memory accesses use the local stack directly.

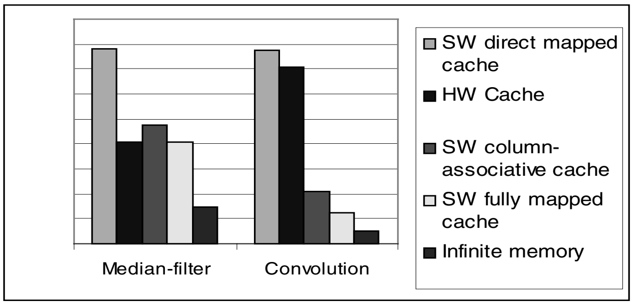


**Fig. 6.** The effect of cache associativity for programs with many conflict misses. Bars shown are normalized to the execution time of the software direct mapped case. For each program, the bars, left to right correspond to (1) a software FlexCache with direct mapping, optimized with Hot Pages, and 256 word cache-lines, (2) A hardware direct mapped cache with 32 word (with 256 word size the performance is worse!) cache-lines, (3) a software FlexCache with 2-way column-associative mapping, optimized with Hot Pages, and 256 word cache-lines, (4) a software FlexCache with fully mapped table, optimized with Hot Pages and 256 word cache lines, and (5) an ideal memory.

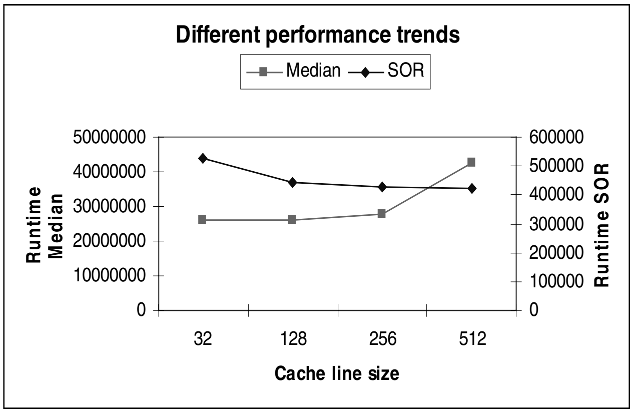


**Fig. 7.** Different line sizes are better for different programs. Runtime for the Median-filter and SOR programs with a software FlexCache using 2-way column associative mapping and optimized with Hot Pages. Median-filter does better with smaller cache lines while the SOR program does better with a larger line size.

without any hardware support, a FlexCache based system can outperform fixed hardware caches by improving caching effectiveness.

We are currently working on evaluating the hardware support in a hybrid FlexCache, and on building low-power *hardware* managed caches that expose just enough of the cache interface to the compiler, such that the FlexCache techniques can be used to reduce tag-array accesses.

## References

1. J. Babb, M. Rinard, C. A. Moritz, M. Frank, W. Lee, R. Barua, and S. Amarasinghe. . In *Parallelizing Applications into Silicon*, Napa, CA, April 1999. IEEE Computer Society.
2. K. D. Cooper and T. J. Harvey. Compiler-Controlled Memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, Oct. 3–7 1998.
3. S. K. R. Erik G. Hallnor. A Fully Associative Software-Managed Cache Design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
4. R. Fromm, D. Patterson, K. Asanovic, A. Brown, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, K. Yelick, T. Anderson, and K. Wawrzynek. Intelligent RAM (IRAM). In *IRAM tutorial, ASP-DAC98*. IEEE Computer Society, Februari 1998.
5. Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrelas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of International Conference on Computer Design (ICCD)*, Oct 1999.

6. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
7. M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, June 1998. IEEE Computer Society.
8. N. P. J. Parthasarathy Ranganathan, Sarita Adve. Reconfigurable Caches and Their Application to Media Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, June 2000. IEEE Computer Society.
9. D. Patterson, T. Andersson, N. Cardwell, R. Fromm, K. Keeton, and C. K. and. A Case for Intelligent RAM: IRAM. *In IEEE Micro*, April 1997.
10. S. A. Rajeev Barua, Walter Lee. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, June 1999. IEEE Computer Society.
11. R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, May 1999.
12. R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 18–21, 1995.

# Aggressive Memory-Aware Compilation*

Peter Grun pgrun@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu
Alex Nicolau nicolau@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA
http://www.cecs.uci.edu/~aces

## Abstract

*Memory delays represent a major bottleneck in embedded systems performance. Newer memory modules exhibiting efficient access modes (e.g., page-, burst-mode) partly alleviate this bottleneck. However, such features cannot be efficiently exploited in processor-based embedded systems without memory-aware compiler support. We describe a memory-aware compiler approach that exploits such efficient memory access modes by extracting accurate timing information, allowing the compiler's scheduler to perform global code reordering to better hide the latency of memory operations. Moreover, we present a compiler technique which in the presence of caches actively manages cache misses, and performs global miss traffic optimizations, to better hide the latency of the memory operations. Our memory-aware compiler scheduled several benchmarks on the TI C6201 processor architecture interfaced with a 2-bank synchronous DRAM and generated average improvements of 24% in the presence of efficient access modes, and 61.6% improvement in the presence of caches, over the best possible schedule using a traditional (memory-transparent) optimizing compiler, demonstrating the utility of our memory-aware compilation approach.*

## 1 Introduction

The memory subsystem is one of the key performance and power bottlenecks in emerging architectures: as the gap widens between processors and memories, long memory latencies hinder processor performance, while simultaneously dissipating more power. Advances in memory technology and memory architectures partially alleviate this problem, for instance through new generations of memories such as SDRAM, and RAMBUS that exhibit efficient access modes (e.g., page-, burst-, and pipelined-access mode). However, these newer memory families still have to be accessed through some form of caching in order to deliver sustained performance. Previous work in optimizing compilers and cache architectures have focused on improving cache performance through exploitation of program locality, program scheduling to hide latencies of cache misses, etc. However, such techniques have traditionally assumed a fairly generic model

*This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship.

of the background memories being accessed. While these newer memory families are designed with special, performance- enhancing access modes, even the most aggressive traditional optimizing compilers are unable to fully exploit such features. Our memory-aware compilation approach explicitly models and captures detailed timing characteristics of newer memory families, and exploits this timing information to further improve processor performance and provide opportunities for power management.

In our approach, we capture the memory access protocols for each memory component through a detailed and accurate timing model for the different memory access modes of a memory component. Using this timing information, our compiler techniques arc able lo belier match the characteristics of the memory sub-system with the specific processor architecture, leading to significant improvements in performance. Traditionally, these access modes were transparent 10 the processor, and were exploited implicitly by the memory controller (e.g., whenever a memory access referenced an element already in the DRAM's row buffer, it avoided the row-decode step, fetching it directly from the row buffer). However, since ihe memory controller only has access to local information, il is unable to perform more global optimizations (such as global code reordering to belter exploit special memory access modes). Our approach provides the compiler with a more accurate timing model for the specific memory access modes, and thus allows our compiler to perform global optimizations that aggressively hide the latency of the memory operations. Moreover, due lo the ubiquity of caches in today's architeclures, optimizing ihe memory accesses in the presence of caches is crucial. In the presence of caches, the accurate timing information allows our compiler to explicitly manage cache miss traffic, generating better performance through the hiding of cache miss latencies.

Our memory-aware compilation approach exploits detailed memory timing information, providing an opportunity to perform global compiler optimizations. First, we extract accurate memory timing from an architectural description of the processor/memory system in the EXPRESSION Architectural Description Language (ADL). Then, we use this detailed memory timing information to efficiently exploit the features of the memory modules, such as page-mode and burst-mode accesses, pipelining and parallelism. Additionally, in cache-based architectures we further improve performance through explicit management of cache miss traffic.

The key idea in our approach is the notion of combining detailed timing of the memory modules (e.g., efficient memory access modes) with the processor pipeline timings to generate accurate operation timings. We then use these exact operation timings to belter schedule ihe application, and hide the latency of the memory operations. Processors traditionally rely on a memory controller lo synchronize and utilize specific access modes of memory modules (e.g., freeze the pipeline when a long delay from a memory read is encountered). However, the memory controller only has a local view of the (already scheduled) code being executed. In the absence of an accurate timing model, the best the compiler can do is to schedule optimistically, assuming ihe fastest access time (e.g., page mode, or a hit in the presence of a cache), and rely on the memory controller to account for longer delays, often resulting in performance penalty. This optimistic approach can be significantly improved by integrating an accurate liming model into the compiler. In our approach, we provide a detailed memory timing model to the compiler so that it can better utilize efficient access modes through global code analysis and optimizations, and help the memory subsystem produce even better performance. We use these accurate operation timings in our retargetable compiler to better hide the latency of the memory operations, and obtain further performance improvements.

Moreover, in the absence of dynamic data hazard detection (e.g., in VLIW processors), these operation timings are *required* to insure correct behavior: the compiler uses them to insert NOPs in the schedule to avoid data hazards. In the absence of a detailed timing model, the compiler is forced to use a pessimistic schedule, thus degrading overall performance.

## 2 Related Work

There has been related work in 2 domains: high-level synthesis and mainstream compilers and architectures. In high-level synthesis. Panda et al. [7] present pre-synthesis optimizations to use the page-mode DRAM access. [6| extend this work to Synchronous and RAMBUS DRAMs, using burst-mode accesses, and exploiting memory bank interleaving.

Recent work on interface synthesis **[1], [2]** present techniques to formally derive node clusters from interface timing diagrams. These techniques can be applied to provide an abstraction of the memory module timings required by our approach.

In [3] and [4| we presented preliminary results on generating and using accurate timing information in the compiler, for page- and burst-mode DRAM accesses, as well as in the presence of caches, hiding the miss latencies by improving the overlap between cache misses and hits to a different cache line.

In the compilers/architectures domain, recent work by Rixner et al. [8] presents a memory controller approach to dynamically reorder the memory accesses, and improve the utilization of the DRAM access modes The dynamic reordering applies only to a window of pending memory accesses. By performing static compiler optimizations, it is possible to further improve the memory access schedule by globally reordering them. We complement this work by making the compiler aware of the memory access modes and timings. Moreover, we apply a similar compiler approach in the presence of caches.

## 3 Overview of Experiments

Our experiments demonstrate the performance gains obtained by using accurate timing in the compiler for the Texas Instruments TIC6201 VLIW DSP [9] processor interfaced with the IBM03I6409C [51 Synchronous DRAM. We first optimize a set of benchmarks to better utilize the efficient memory access modes (e.g., through memory mapping, code reordering or loop unrolling), and then we use the accurate timing model to further improve the performance by hiding the latencies of the memory operations. To separate out the benefit of the better timing model from the gain due to the access mode optimizations and the access modes themselves, we present the set of results which show the performance gains obtained by scheduling with accurate timing in the presence of a code already optimized for memory accesses, and compare them to the performance of the same memory-access-optimized code using less accurate timing, scheduled optimistically, assuming the shortest access time available (page-mode access), and relying on the memory controller to account for longer delays. This optimistic scheduling is the best alternative available to the compiler, short of an accurate timing model. The performance gains from exploiting detailed memory timing vary from 6% to 47.9%, and an average of 23.9% over a schedule that exploits the efficient access modes without detailed timing. We also compare the above approaches to the baseline performance of the system in the absence of efficient memory access modes.

Our second set of experiments demonstrate the performance gains obtained by aggressively optimizing the memory miss traffic on a set of multimedia and DSP benchmarks. We perform the optimization in two phases: first we isolate the cache misses and attach accurate hit and miss timing to the memory accesses, to allow the scheduler to better target the memory subsystem architecture, obtaining between 15.2% and 52.8% performance improvement over the traditional compiler. We then further optimize the cache miss traffic, by loop shifting to reduce the intra-iteration dependence chains due lo accesses to the same cache line, and allow more overlap between memory accesses, resulting in a further 21.3% average performance improvement.

Currently, our work applies to wide issue statically scheduled VLIW Processors, and preliminary results have been presented at DAC-2000 [3] and ICCAD-2000 [4]. We believe that our techniques are also applicable to dynamically scheduled processors. Our on-going work evaluates the improvements of our approach for out-of-order issue superscalar processors, and also addresses the tradeoff between increase in code size (due to loop unrolling) versus performance improvement.

## References

[1] P. Chou, R. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD,* 1995.

(2] K.-S. Chung. R. Gupta, and C. L. Liu. Interface co-synihesis techniques for embedded systems. In *ICCAD,* 1996.

[3] P. Grun, N. Dutt, and A. Nicolau. Memory aware compilation through accurate timing extraction. In *DAC.* 2000.

[4] P. Grun, N. Dutt, and A. Nicolau. Mist: An algorithm for memory miss traffic mangement. In *ICCAD.* 2000.

[51 IBM Microelectronics. Data Shocls for Synchronous DRAM 1BM03I6409C. *www. ch ips. ibm. com/products/memoryA)8J3348/.*

|6] A. Khare. P. R. Panda. N. D- Dun, and A. Nicolau. High level synthesis with synchronous and rambus drams. In *SASIMI,* Japan. 1998.

[7] P. Panda. N. Dutl. and A. Nicolau. *Memory Issues in Embedded Syslems-on-Chip.* Kluwcr, 1999.

[8J S. Rixner, W. Dally. U. Kapasi. P. Matlson, and J. Owens. Memory access scheduling. In *ISCA,* 2000.

19) Texas Instruments. *TMS32OC620I CPU and Instruction Set Reference Guide.*

# Energy/Performance Design of Memory Hierarchies for Processor-in-Memory Chips[⋆]

**Michael Huang**[†], **Jose Renau**[†], **Seung-Moon Yoo** [‡], **and Josep Torrellas**[†]
[†]Department of Computer Science
[‡]Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{weihuang,renau,yoo2,torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/flexram

## 1 Introduction

Merging processors and memory into a single chip has the well-known benefits of allowing high-bandwidth and low-latency communication between processor and memory, and reducing energy consumption. As a result, many different systems based on what has been called Processor In Memory (PIM) architectures have been proposed [1, 3, 7, 8, 10, 12–16, 18].

Recent advances in technology [4, 5] appear to make it possible to integrate logic that cycles nearly as fast as in a logic-only chip. As a result, processors are likely to put much pressure on the relatively slow on-chip DRAM. To handle the speed mismatch between processors and DRAM, these chips are likely to include non-trivial memory hierarchies in each DRAM bank.

With many on-chip high-frequency processors, all of them potentially accessing the memory system concurrently, these chips will consume much energy. In addition, these chips are likely to be used in non-traditional places like the memory of a server [3, 7, 12] or the I/O subsystem [1], which may not have heavy-duty cooling support. Consequently, it is important to design the chips for energy efficiency.

In this abstract, we examine, from a performance and energy-efficiency point of view, the design of the memory hierarchy in a multi-banked PIM chip with many simple, fast processors. Our results suggest the use of per-processor memory hierarchies that include modest-sized caches, simple DRAM bank organizations that support segmentation, and no prefetching.

## 2 Memory Hierarchies for PIM Chips

Our focus architecture is a PIM chip that includes tens of relatively simple, high-frequency processors, each of which is associated with a bank of DRAM. Such a design has been suggested for systems like Active Pages [12, 13], FlexRAM [7], and DIVA [3] among others. The chip can be modeled as in Figure 1-(a), where the organization of the

[⋆] This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, and gifts from IBM and Intel.

processors, memory, and network may vary. We feel, however, that currently-proposed designs are relatively conservative in logic speed. Recent advances in technology appear to allow logic to cycle nearly as fast as in a logic-only chip [4, 5]. This means that these chips may soon include processors cycling at about 800-1000 MHz. Such processors are likely to put much pressure on the slower DRAM.

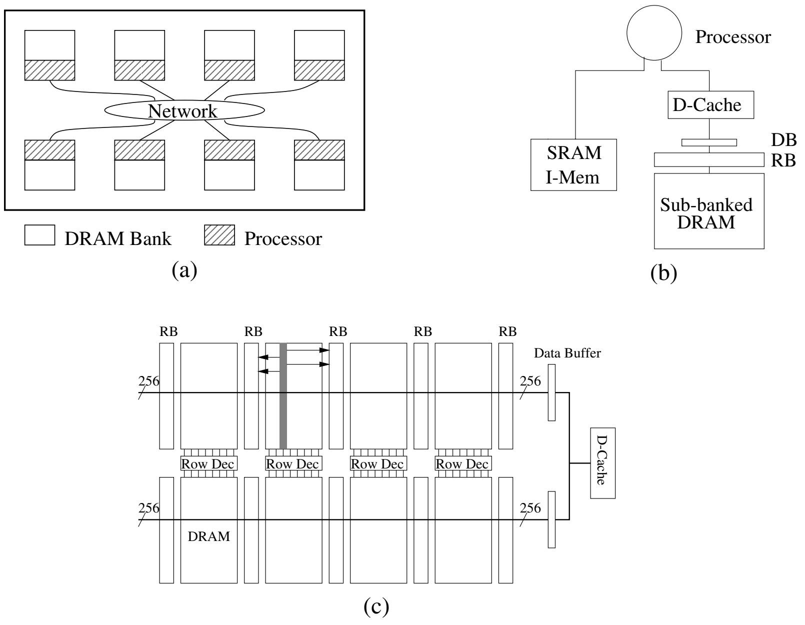


**Fig. 1.** Example of chip architecture considered. *RB*, *DB*, and *Row Dec* stand for row buffer, data buffer and row decoder, respectively.

To handle the speed mismatch between processors and DRAM, these chips are likely to associate a non-trivial memory hierarchy to each DRAM bank. In this paper, we assume a per-bank baseline memory hierarchy as in Figure 1-(b). In the figure, the instruction memory hierarchy includes a fast SRAM memory. The data memory hierarchy includes a cache with hardware sequential prefetch of 1 line. The DRAM bank itself is sub-banked and has row and data buffers. For example, Figure 1-(c) shows the DRAM organized into 8 sub-banks, with 10 row buffers, and 2 256-bit data buffers.

Unlike in memory-only chips, where the DRAM organization is often limited to standard designs, embedded systems allow many different organizations for the DRAM array. For example, designers can change the width and length of a DRAM sub-bank, and the number of sub-banks. These changes can affect the performance delivered and the energy consumed by DRAM accesses, and the area utilized.

In a traditional DRAM array organization, when a bank is accessed, every other sub-bank is activated. Consecutive sub-banks are not activated because they share a row buffer. Figure 2-(a) shows a 4 sub-bank organization. We now consider three improvements: segmentation, interleaving, and pipelining.

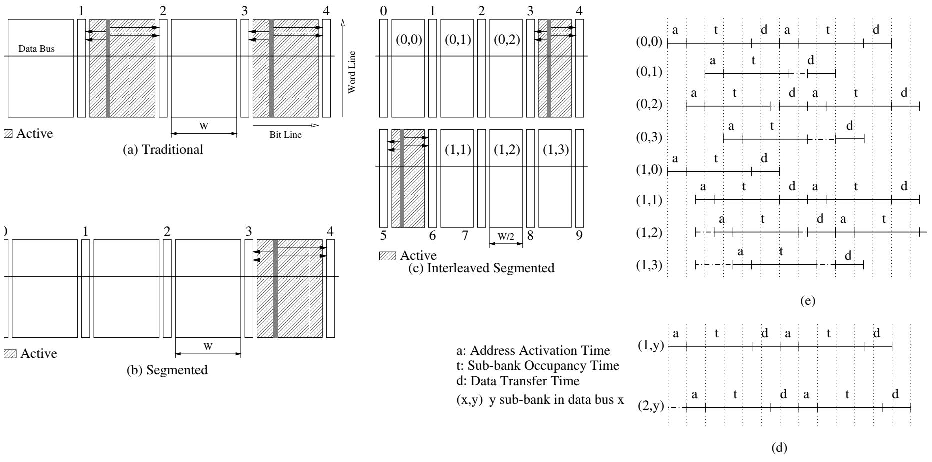

Data Bus
Word Line
Bit Line
W
▨ Active
(a) Traditional
(b) Segmented
W/2
(c) Interleaved Segmented
a: Address Activation Time
t: Sub-bank Occupancy Time
d: Data Transfer Time
(x,y) y sub-bank in data bus x
(d)
(e)

**Fig. 2.** Different DRAM bank organizations and timings.

With segmentation (Figure 2-(b)), only one sub-bank is activated at a time. The resulting row buffer decoupling changes the hit rate of the row buffers. In addition, DRAM accesses consume less energy: because only half of the bit lines are activated, about 50% of the energy is saved.

With interleaving, each sub-bank is vertically sliced and a data bus is assigned to each of the resulting slices. Figure 2-(c) shows a 2-way interleaved system. The performance is higher because both data busses work in parallel (Figure 2-(d) shows a timing diagram with the maximum overlap, assuming a single address bus). As for energy, although row buffer hits now cost a bit more, DRAM accesses again save about 50% of the energy because only half of the cells are activated. The area used increases.

Finally, one problem shown in Figure 2-(d) is that reads from different sub-banks that share a data bus are serialized by long sub-bank occupancy times. With pipelining, these sub-banks can overlap their occupancy times (Figure 2-(e)). The only serialization happens in the shared address bus and data bus. The result is higher performance. As for energy, pipelining has only a small impact.

## 3 Evaluation Environment

We evaluate the PIM chip of Section 2 using a MINT-based simulation system [9]. The architecture modeled is a single chip with 64 processors connected in a ring. Each processor is associated with a 1-Mbyte DRAM bank like in Figure 1-(b). The baseline parameters of each processor-bank pair are shown in Table 1. The target technology is IBM's 0.18 $\mu$m Blue Logic SA-27E ASIC [4] with the default voltage of 1.8 V.

| Processor | D-Cache | I-Memory | Data Buffer | Row Buffer | Sub-Bank |
|---|---|---|---|---|---|
| 2-issue in-order 800MHz | Sz: 8KB, WB | Size: 4 Kinst. | Number: 1 | Number: 5 | Number: 4 |
| BR Penalty: 2 cycles | Assoc: 2 | Line: 4 inst. | Size: 256 b | Size: 1 KB | Cols: 4096 |
| Int,Ld/St,FP Units: 2,1,0 | Line: 32 B | RTrip:1.25ns | Bus: 256 b | Bus: 256 b | Rows: 512 |
| Pending Ld,St: 2,2 | RTrip:1.25ns | | RTrip:3.75ns | RTrip:7.5ns | RTrip:15 ns |

**Table 1.** Parameters for a single memory bank and processor pair. In the table, *BR* and *RTrip* stand for branch and contention-free round-trip latency from the processor, respectively.

| Appl. | What It Does | Problem Size | D-Cache Hit Rate | Average Power(W) |
|---|---|---|---|---|
| *GTree* | Data mining: tree generation | 5 MB database, 77.9 K records, 29 attributes/record | 50.7% | 10.2 |
| *DTree* | Data mining: tree deployment | 1.5 MB database, 17.4 K records, 29 attributes/record | 98.6% | 10.8 |
| *BSOM* | BSOM neural network | 2 K entries, 104 dims, 2 iters, 16-node network, 832 KB db | 94.7% | 15.5 |
| *BLAST* | BLAST protein matching | 12.3 K sequences, 4.1 MB total, 1 query of 317 bytes | 96.9% | 8.7 |
| *Mpeg* | MPEG-2 motion estimation | 1 1024x256 frame plus a reference frame. Total 512 KB | 99.9% | 11.3 |
| *FIC* | Fractal image compressor | 1 512x512 image, 4 512x512 internal structure. Total 2 MB | 97.8% | 6.1 |

**Table 2.** Applications executed.

The names for the DRAM bank organizations that we evaluate are *Trad*, *S*, *SP*, *IS*, and *ISP*, which refer to traditional, segmented, segmented pipelined, interleaved segmented, and interleaved segmented pipelined, respectively. In each case, we add $(i, j)$ to refer to $i$-ways interleaved with $j$ sub-banks per way.

To estimate the energy consumed in the chip, we have applied scaling-down theory to data on existing devices reported in the literature, as well as used several techniques and formulas reported in the literature [6, 17, 19, 20]. We add the contributions of the processors, clock, memory hierarchies, and other modules. A detailed discussion of the methods that we have followed can be found in [21]. In [21], we have additionally validated our estimates with CACTI [19] and with published results on the ARM processor [11].

For the experiments, we use 6 applications that are suitable to the integer-based PIM chip considered: they access a large memory size, are very parallel, and are integer based. They come from several industrial sources. We have parallelized each application into 64 threads by hand.

Table 2 lists the applications and their characteristics. They include the domains of data mining, neural networks, protein matching, multimedia, and image compression. Each application runs for several billions of instructions.

## 4 Evaluation

The best memory hierarchy organization depends on the metric being optimized. We consider two metrics: performance and energy-delay product. In our evaluation, we start with the baseline architecture of Section 3 and then vary it. As a reference, we use an ideal architecture (*Perf*): loads and stores are satisfied with zero latency and consume no energy in the memory system.

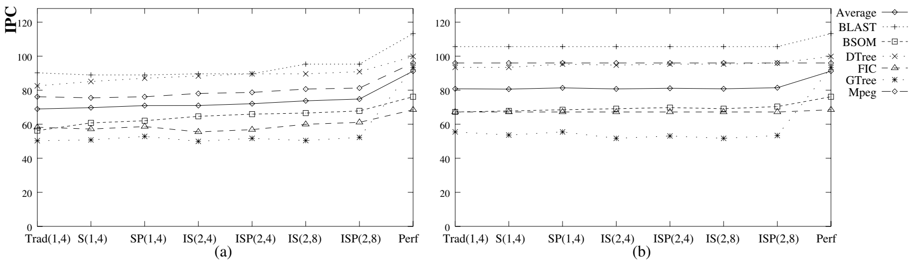


**Fig. 3.** Effect of the DRAM bank organization on the IPC in systems with 1-Kbyte (a) and 8-Kbyte (b) data caches.

### Maximizing Performance

To compare performance, we measure the average IPC delivered by the combined 64 processors for the duration of the application. We first evaluate the effect of the memory bank organization. Figure 3 shows the IPC of the applications running on the baseline architecture for different memory bank organizations. Charts (a) and (b) correspond to systems with 1- and 8-Kbyte D-caches, respectively. The memory organizations are ordered from the simpler ones on the left side to the more sophisticated ones on the right side. Each chart has an *Average* line that tracks the average of all applications.

Figure 3-(a) shows that performance improves slightly as we move to the more sophisticated designs. Going from *Trad(1,4)* to *ISP(2,8)* increases the IPC by an average of 8%. However, for 8-Kbyte caches (Figure 3-(b)), the changes are very small. This is because, with large caches, there are relatively few cache misses and, as a result, the type of DRAM bank organization matters less.

Comparing the IPC in *Perf* and *ISP(2,8)*, we see the IPC lost in the most advanced memory system. This fraction is on average 18% and 11% in Figures 3-(a) and (b).

Figure 5-(a) shows the effect of the cache size and prefetching support. We consider the baseline architecture with three different DRAM bank organizations: conservative (*Trad(1,4)*), aggressive (*ISP(2,8)*), and in-between (*IS(2,4)*). The figure shows the IPC averaged over all applications. We analyze caches of 256 bytes, 1 Kbyte, 8 Kbytes, and 16 Kbytes, all with and without prefetching. For each memory organization, there are 8 bars, labeled with the cache size in bytes followed by *P* or *NP* for prefetching or not prefetching, respectively.

The best performance is achieved with the largest cache size (16 Kbytes). However, large caches deliver diminishing returns. The figure also shows that adding the simple prefetching support considered here makes little difference to performance.

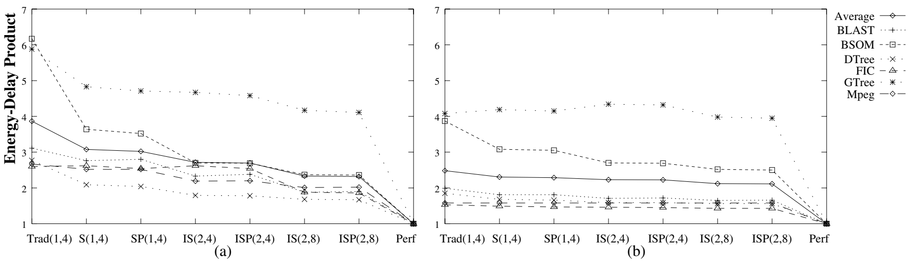


**Fig. 4.** Effect of the DRAM bank organization on the energy-delay product in systems with 1-Kbyte (a) and 8-Kbyte (b) data caches.

## Minimizing the Energy-Delay Product

In embedded systems, a common figure of merit is the energy-delay product [2]. A low product implies that the system is both fast and energy-efficient. Consequently, in this section, we compare the energy-delay product of the chips with different memory hierarchy designs. To compute the energy consumed, we add up the contributions of all the subsystems in the chip.

Figures 4-(a) and 4-(b) show the energy-delay product of the chip under the baseline architecture for different DRAM bank organizations. Charts (a) and (b) correspond to systems with 1- and 8-Kbyte D-caches respectively, and are organized as in Figures 3-(a) and 3-(b). For each application, the charts are normalized to *Perf*.

In systems with 1-Kbyte caches (Figure 4-(a)), the average energy-delay product decreases for the more advanced memory organizations. For example, the product in *ISP(2,8)* is only 60% of that in *Trad(1,4)*. The reason is that advanced DRAM bank organizations deliver slightly higher IPCs and consume much less energy in the process. However, as caches increase to 8 Kbytes (Figure 4-(b)), the changes are smaller. Overall, for 8-Kbyte cache systems, only segmentation (going from *Trad(1,4)* to *S(1,4)*) makes a significant difference. Supporting interleaving and increasing the number of sub-banks from (2,4) to (2,8) has only a small effect.

Figure 5-(b) measures the energy-delay product for the average of all applications for different cache sizes and prefetching support. The bars are normalized to *Perf*. From the figure, we see that designs with larger caches tend to have lower energy-delay products. For example, in *Trad(1,4)*, the product with 16-Kbyte caches is about 30% of the product with 256-byte caches. The reason is that caches have a double effect: they speed up the program and, in addition, eliminate energy-consuming memory accesses. We observe, however, that for the more advanced memory organizations and large caches, the trend reverses: 16-Kbyte caches are slightly worse than 8-Kbyte caches. The reason is

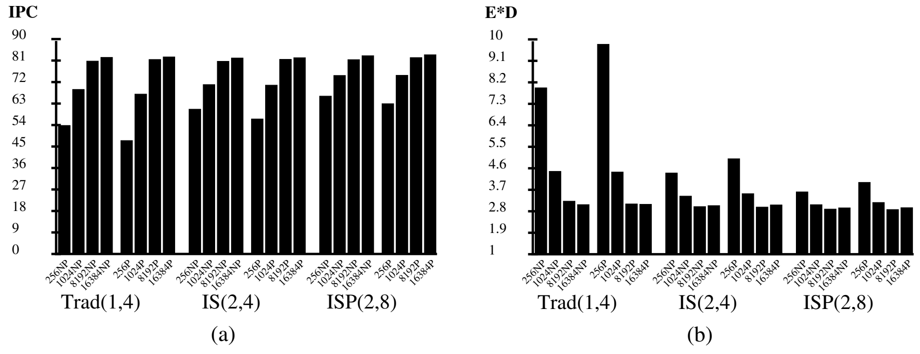


**Fig. 5.** Effect of the cache size and prefetching support on IPC (a) and energy-delay product (b).

that the diminishing returns in lower miss rates delivered by larger caches do not compensate for the higher energy consumption that larger caches require. We also see that simple prefetching does not help.

## 5 Discussion

In a PIM chip like the one analyzed here, minimizing the energy-delay product is likely to be the top priority. Our results suggest to use modest-sized D-caches (8 Kbytes), a simple DRAM bank organization that supports only segmentation, and no prefetching. Modest-sized caches are effective: they speed-up the application, are energy-efficient, consume modest area, and render fancy DRAM bank organizations largely unnecessary. If area is not an issue, the energy-delay product can be improved slightly by supporting interleaving in the DRAM bank and increasing the number of sub-banks.

## References

1. A. Brown et al. ISTORE: Introspective Storage for Data-Intensive Network Services. *Workshop on Hot Topics in Operating Systems*, March 1999.
2. R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal on Solid-State Circuits*, 31(4):1277–1284, September 1996.
3. M. Hall et al. Mapping Irregular Aplications to DIVA, a PIM-Based Data-Intensive Architecture. In *Supercomputing*, November 1999.
4. IBM Microelectronics. Blue Logic SA-27E ASIC. http://www.chips.ibm.com/news/1999/sa27e, February 1999.
5. S. Iyer and H. Kalter. Embedded DRAM Technology: Opportunities and Challenges. *IEEE Spectrum*, April 1999.
6. M. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *International Symposium on Low Power Electronics and Design*, pages 143–148, 1997.
7. Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, October 1999.

8. P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Frontiers of Massively Parallel Computation Symposium*, 1996.
9. V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.
10. K. Mai et al. Smart Memories: A Modular Reconfigurable Architecture. In *International Symposium on Computer Architecture*, June 2000.
11. J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid State Circuits*, 31(11):1703–1714, November 1996.
12. M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture*, pages 192–203, June 1998.
13. M. Oskin et al. Exploiting ILP in Page-Based Intelligent Memory. In *International Symposium on Microarchitecture*, 1999.
14. D. Patterson et al. A Case for Intelligent DRAM. *IEEE Micro*, pages 33–44, 1997.
15. D. Patterson and M. Smith. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. 1997.
16. S. Rixner et al. A Bandwidth-Efficient Architecture for Media Processing. In *International Symposium on Microarchitecture*, November 1998.
17. C-L. Su and A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. In *International Symposium on Low Power Electronics and Design*, pages 63–68, April 1995.
18. E. Waingold et al. Baring It All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.
19. S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.
20. N. Yeung et al. The Design of a 55SPECint92 RISC Processor under 2W. *ISSCC Digest of Technical Papers*, pages 206–207, February 1994.
21. S-M. Yoo, J. Renau, M. Huang, and J. Torrellas. FlexRAM Architecture Design Parameters. Technical Report CSRD-1584, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2000. http://iacoma.cs.uiuc.edu/flexram/publications.html.

# SAGE: A New Analysis and Optimization System for FlexRAM Architecture

Tsung-Chuan Huang and Slo-Li Chu

Department of Electrical Engineering, National Sun Yat-sen University
Kaohsiung, Taiwan, R.O.C.
tch@mail.nsysu.edu.tw   d8631817@student.nsysu.edu.tw

**Abstract.** Intelligent memory is a new class of computer architecture, to reduce the performance gap between the processor and memory. After analyzing a region of application, we decide to take "statement" viewpoint to extract more potential benefit of program running on intelligent memory architecture. Then we develop our SAGE system, a "statement" base analysis system, different from other iteration base system. In this paper, we will describe how SAGE split statement and make an acceptable schedule to execute on PHost and PMem simultaneously. Finally we will discuss our recently result of this approach.

## 1 Introduction

In order to solve the performance gap between the processor and memory and to exploit maximum memory bandwidth, many researchers proposed a new class of computer architecture: Intelligent Memory [2][6][7][8]. After comparing several intelligent memory systems, we decide to adopt UIUC's FlexRAM [3,10] architecture as our basic platform to develop optimization and programming model.

When examining current Intelligent Memory systems, we find that they usually focus on taking fully parallel applications as their benchmarks and spawning a lot of PIM processors, like massive multiprocessor systems, to exploit more parallelism. However, these results cannot reveal the real benefit of Intelligent Memory. Therefore we try to explore another research direction: to improve the performance of general application, instead of fully parallelizable ones. Here, we propose our SAGE (Statement-Analysis-Grouping-Evaluation) system, a new analysis model with a suite of optimizing skills, to extract parallelizable portion of general programs and to achieve better load balance between PHost and PMem.

In our recent experimental results, quite good speedup is obtained, which exceeds the limitation of computation capability of PMem, in one-PHost-one-PMem environment. In what follows, we will briefly describe analysis stages of SAGE first, then present algorithms and an example to demonstrate how it works. Finally, we will discuss our experimental results of synthetic example and two real benchmark programs in SPEC 95 and BLAS3.

## 2 Intelligent Memory Architecture

A general view of the FlexRAM [3,10] architecture is shown in Figure 1. Each FlexRAM chip has 1 PMem and 64 PArray memory processors. The host processor of the target workstation is called PHost. The architecture parameters are listed in Table 1.

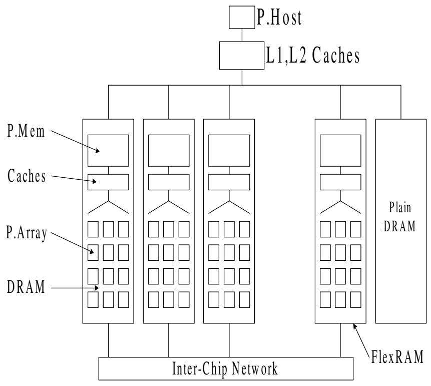


**Fig. 1**. The organization of FlexRAM architecture.

**Table 1**. Parameters of the FlexRAM architecture.

| P.Host | P.Host L1 & L2 | Bus & Memory |
|---|---|---|
| Freq: 800 MHz<br>Issue Width: 6<br>Dyn Issue: Yes<br>I-Window Size: 96<br>Ld/St Units: 2<br>Int Units: 6<br>FP Units: 4<br>Pending Ld/St: 8/8<br>*BR* Penalty: 4 cyc | L1 Size: 32 KB<br>L1 *RT*: 2.5 ns<br>L1 Assoc: 2<br>L1 Line: 64 B<br>L2 Size: 256 KB<br>L2 *RT*: 12.5 ns<br>L2 Assoc: 4<br>L2 Line: 64 B | Bus: Split Trans<br>Bus Width: 16 B<br>Bus Freq: 100 MHz<br>PHost Mem *RT*:<br>262.5 ns<br>PMem Mem *RT*:<br>50.5 ns |
| P.Mem | P.Mem L1 | P.Array |
| Freq: 400 MHz<br>Issue Width: 2<br>Dyn Issue: No<br>Ld/St Units: 2<br>Int Units: 2<br>FP Units: 2<br>Pending Ld/St: 8/8<br>*BR* Penalty: 2 cyc | L1 Size: 16 KB<br>L1 *RT*: 2.5 ns<br>L1 Assoc: 2<br>L1 Line: 32 B<br>L2 Cache: No | Freq: 400 MHz<br>Issue Width: 1<br>Dyn Issue: No<br>Pending St: 1<br>Row Buffers: 3<br>*RB* Size: 2 KB<br>*RB* Hit: 10 ns<br>*RB* Miss: 20 ns<br>*RB* Penalty: 2 cyc |

* BR stands for branch, RT for round-trip latency from the processor, and RB for row buffer.

In order to simplify the problem and exploit the benefits of PIM architectures, in this paper, we only consider the system with a single host processor (PHost) and a single memory processor (PMem). We are in process of extending our SAGE system to fit more complicated PIM architectures.

## 3 System Organization

The organization of SAGE is shown in Figure 2. This system will provide four major advantages. First, instead of iteration, SAGE adopts simple-statement loop as the basic execution unit. This different approach will provide a novel methodology to find better schedule in procedure-level parallelism. Second, for its simple, flexible optimizing stages, SAGE can cooperate with other traditional iteration-base analysis systems (such as UIUC’s Polaris) to explore the potential parallelism easily. Third, from its heuristic scheduling mechanism, SAGE can generate suitable execution procedures and dispatch them to PHost and PMem in accordance with practical system configuration. Fourth, programmers can use this analysis model easily to develop their applications for FlexRAM or restructure the original sequential (or parallel) program into a load-balanced, task- separated form. The following of this section we will describe some important stages of this system.

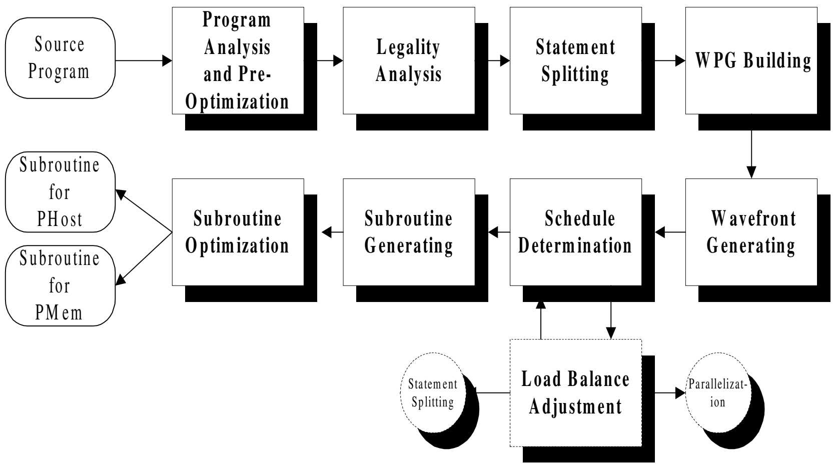


**Fig. 2.** Analysis and optimization stages of SAGE system.

## 3.1 Statement Splitting

In this stage, we use *Loop Distribution* [1][4] ,a well-known transformation of parallelizing techniques, to split original dependence graph by construct our *Weighted Partition Dependence Graph (WPG)*. Then it will be applied in our following optimization and scheduling stages.

**Definition 1 (Loop Denotation) [4]**

A loop is denoted by L = ( $I_1, I_2, \ldots\ldots.. I_d$ )( $S_1, S_2, \ldots\ldots.. S_k$ ), where $I_j$ is a loop index, and $S_j$ is a body statement which maybe an assignment statement or another loop.

**Definition 2 ( Node Partition $\Pi$ ) [4]**

On the dependence graph G , for a given loop L, we define a node partition $\Pi$ of { $S_1, S_2, \ldots\ldots.. S_d$ } in such a way that $S_k$ and $S_l$ , $k \neq l$ , are in the same subset if and only if $S_k \Delta S_l$ and $S_l \Delta S_k$ , where $\Delta$ is an indirectly data dependent relation. On the partition $\Pi$={ $\pi_1, \pi_2$ }, we defined partial ordering relations $\alpha$ , $\overline{\alpha}$ , and $\alpha^o$ , such that for i$\neq$j

1) $\pi_i \ \alpha \ \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta S_l$ , where $\delta$ is true dependence relation .

2) $\pi_i \overline{\alpha} \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \overline{\delta} S_l$ where $\overline{\delta}$ is anti dependence relation.
3) $\pi_i \alpha^o \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta^0 S_l$ where $\delta^0$ is output dependence relation.

**Definition 3 (Weighted Partition Dependence Graph)**

For a given node partition $\Pi$ as in Definition 2, we define a weighted partition dependence graph *WPG(P,E)*. In *WPG* node $p_i \in P$, $p_i(I_i, S_i, W_i, O_i)$ represents a partition $\pi_i \in \Pi$ where $I_i, S_i$ are the same as Definition 1, $W_i$ *(PH,PM)* denotes the PHost/PMem weight value of this node , $O_i$ denotes the execution order of this node.

**Algorithm 1. (Statement Splitting Algorithm)**

Given a loop L = $(I_1, I_2, \ldots\ldots I_d)(S_1, S_2, \ldots\ldots S_d)$

```
Step1: By analyzing subscript expressions and indexing pattern construct a
       dependence Graph G
Step2: On G establish a node partition $\Pi$ as in Definition 2. If there are some large
       partition caused by control dependent relations, we use some approach to
       transfer control dependence to data dependence[5] and then partitioning it
       as above.
Step3: On the partition $\Pi$ establish a weighted partition dependence graph
       WPG(P,E) as in Definition 3
```

### 3.2 Wavefront Generating and Scheduling

In this section, we propose an algorithm for the scheduling of PHost and PMem. In our method, the weights of the blocks in partition P are determined first, then the execution order for each block is determined according to their dependence relation and lexicographic order. The blocks that can be executed simultaneously are assigned in a wavefront. The blocks in the same wavefront are scheduled on PHost and PMem processors based on the weights of them.

## 4 Example

To illustrate how to analyze and optimize program by SAGE system, we take a simple program as example, which is shown in Figure 3. Assume that this program contains three major loops:

**Algorithm 2. (Weighting and Scheduling Algorithm)**

**[Input]**
*WPG=(P(I,S,W,O),E)*, the original weighted partition dependence graph without weight $W$ *(PH,PM)* and order $O$ assignment.

**[Output]**
A Execution wavefront schedule $Wf = \{Wf_1, Wf_2 ....\}$ where $Wf_i = \{PH(P_i...P_j), PM(P_k.....P_l)\}$, $PH(P_i....P_j)$ denoted that partitions for PHost execute in wavefront *i*, $W$ $_i$ denoted that partitions for PMem execute in wavefront *i*.

**[Intermediate]**
W is a working set of nodes ready to visit
max_wf is the maximum number of wavefront
max_pred_O( $p_i$ ) is maximum execution order value $O_i$ of all $p_i$ 's predecessor partitions
PHW( $p_i$ ) is PHost weight value of $p_i$
PMW( $p_i$ ) is PMem weight value of $p_i$

**[Algorithm]**

```
/*Initialization and identifing weight */

  for each $i \ni p_i \in P$ do begin
      $W_i$ (PH,PM)= determine_weight ( $I_i$ , $S_i$ )
      $O_i = 0$
  end for

/* Determining Execution Order */

for each $i \ni p_i$ that has no predecessors do begin
   $O_i = 1$
   W=W-{ $p_i$ }
end for
done = False
```

```
  while done = False AND W $s_2$ $\phi$ do begin
     done=True
     for each $i \ni p_i \in$ W do begin
       if max_pred_O( $p_i$ )=0  then
          done=False
       else
          $O_i$ = max_pred_O( $p_i$ )+1
          W=W-{ $p_i$ }
          max_wf=$O_i$
       end if
     end for
 end while

/*Scheduling*/

wf=1 /* current wavefront number */

for wf=1 to max_wf
   pick all $i \ni O_i$ =wf , store all $p_i$ in wf_tmp
     while done = False do begin
        done = False
        divide wf_tmp into two arbitrary subsets a, b
           /* here $a \cup b$ =wf_tmp. $a \cap b = \phi$ */
           if $|PHW(a) - PHW(b)|$ is minimal of all possible a, b . then
                $Wf_{(wf)} = \{PH(a), PH(b)\}$
                done=Ture
           end if
     end while
end for
```

Firstly, the preliminary analysis and pre-optimization are applied, then statements are separated. According to the results of legality analysis, we can distinguish which statement can be split. The statement unable to be split will keep the original form.. All other statements will be split into a single-statement loop. Figure 4. shows the results of loop 2 after statement splitting.

```
      DO I = 1 TO N
         DO J = 1 TO M
S1:         A(I,J) = B(I,J)+C(I,J)
S2:         A(I,J) = A(I-1,J)+A(I+1,J)+C
S3:         X = A(I,J)+2
S4:         Y = X*C
S5:         D(I,J) = 2*D(I,J)+3
S6:         E(I,J) = 2*E(I,J)+2
         ENDDO
      ENDDO
```

**(a) Loop 1**

```
      DO J = 1 TO N
         DO I = 1 TO M
S7:         F(I,J) = E(I,J)*F(I,J)
S8:         F(I,J+1) = F(I,J)+5
S9:         G(I,J) = G(I-1,J)*G(I,J-1)
         ENDDO
      ENDDO
```

**(b) Loop 2**

```
      DO I = 1 TO N
         DO J = 1 TO M
S10:        Z = A(I,J)+A(I,J-1)
S11:        A(I,J) = Z*C
         ENDDO
      ENDDO
```

**(c) Loop 3**

**Fig. 3.** A simple program with three individual loops.

```
      DO J = 1 TO N
         DO I = 1 TO M
S7:         F(I,J) = E(I,J)*F(I,J)     •     b4
         ENDDO
      ENDDO

      DO J = 1 TO N
         DO I = 1 TO M
S8:         F(I,J+1) = F(I,J)+5        •     b5
         ENDDO
      ENDDO

      DO J = 1 TO N
         DO I = 1 TO M
S9:         G(I,J) = G(I-1,J)*G(I,J-1) •     b6
         ENDDO
      ENDDO
```

**Fig. 4.** Result of loop 2 after statement splitting.

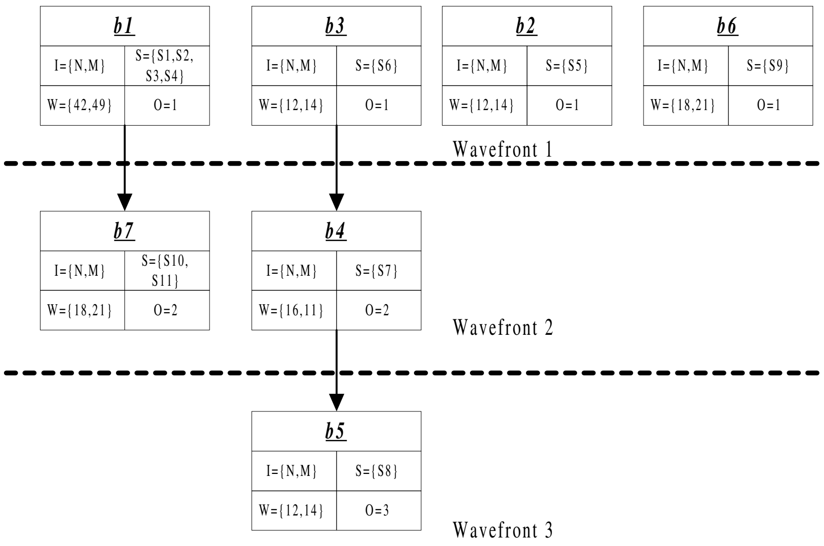


**Fig. 5.** WPG graph before SAGE generates execution schedule.

The Weighted Partition Dependence Graph (WPG) can be drawn as in Figure 5. After the manipulation of scheduling stage (cf. Algorithm 2), the system will generate the good execution schedule: $WF_{(1)}$ = {PH (b1) ,PM(b2, b3, b6)}, $WF_{(2)}$={PH(b4), PM(b7)}, $WF_{(3)}$={PH(b5)} using Algorithm 3 according to the PHost and PMem's computation power and characteristics.

## 5 Experimental Results

The code generated by SAGE is targeted to FlexRAM simulator [3,10] developed by IA-COMA Lab in UIUC. This simulation environment models dynamic superscalar multiprocessor and detailed memory behaviors cycle by cycle. The detailed configuration is shown in Table 1. In addition to the original PHost processor, we only spawn one PMem processor. In order to reflect the advantages of FlexRAM, we also experiment on the general heterogeneous environments with two processors, in which the co-processor is identical to PMem other than its weak memory access capability (because it is not in the memory).

The applications evaluated include four programs: *swim* is from SPEC95, *strmm* is from BLAS3, and *example* is the synthetic program we proposed in this paper. Table 2 shows the execution time for these three applications.

The approximated performance ratio between PHost and PMem is 8:1. This means that if PMem works with PHost simultaneously, the speedup will be up to 1.125 theoretically. But we get more than 1.125 in Table 2. The reason to explain this result is that PMem has shorter memory access latency. This attests the major objective of Intelligent Memory system: to reduce the performance gap between processor and memory.

**Table 2**. Performance results of three programs: swim (SPEC95), strmm (BLAS3), and our demonstration example.

| ***Benchmarks*** | ***P.Host Exec. Cycles*** | ***Optimized Cycles*** | ***Speed UP /P.Host*** |
|---|---|---|---|
| ***swim*** | 86144754 | 64190746 | 1.342 |
| ***hstrmm*** | 107235775 | 76868804 | 1.395 |
| ***example*** | 10129486 | 6889560 | 1.470 |

## 6 Conclusion

In this paper, we propose statement spitting and scheduling mechanisms for intelligent memory architectures to exploit the computing power of the host and memory processors. The algorithms are general enough, not only for FlexRAM architecture but also can be applied on heterogeneous mix of processors systems. We hope this system will provide a reference for other researchers who intend to develop new optimization and parallelization techniques for Intelligent Memory.

## References

1. Allen, J. R., Callahan, D., and Kennedy, K.: Automatic decomposition of scientific programs for parallel execution. In Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages, Munich, Germany, Jan. (1987)
2. Granacki, J. et al.: Data Intensive Architecture: DIVA. http://www.isi.edu/asd/diva/, (1998)
3. Huang, W.: Exploiting Application Parallelism Using Advanced Intelligent Memory – The FlexRAM approach. Ms Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (1999)
4. Kuck, D.j.: A survey of parallel mechine organization and programming. ACM Comput. Surv. 9, 1, Mar. (1977), 29-59
5. Kennedy, K., and McKinley, K. S.: Loop distribution with arbitrary control flow. In Proceedings of Supercomputing '90,New York, NY, Nov. (1990)
6. Kogge, P.: The EXECUBE Approach to Massively Parallel Processing. In proceedings of the 1994 International Conference on Parallel Processing, August. (1994)
7. Oskin, M., Chong, F., and Sherwood, T.: Active Pages: A Computation Model for Intelligent Memory. In Proceedings of the 25th Annual International Symposium on Computer Architecture, pages, June. (1998), 192-203
8. Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Tomas, R., and Yelick, K.: A Case for Intelligent DRAM. In IEEE Micro, March/April (1997), 33-44
9. Veenstra, J., and Fowler, R.: MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In MAS-COTS'94, January. (1994), 201-207
10. Kang Y., Huang W., Yoo S., Keen D., Ge Z., Lam V., Pattnaik P., and Torrellas J.: FlexRAM: Toward an Advanced Intelligent Memory System, International Conference on Computer Design (ICCD), Austin, Texas, Oct. 1999.

# Performance/Energy Efficiency of Variable Line-Size Caches for Intelligent Memory Systems

Koji Inoue[1,2], Koji Kai[1], and Kazuaki Murakami[3]

[1] Institute of Systems & Information Technologies/KYUSHU, 2-1-22 Momochihama, Sawara-ku, Fukuoka 814-0001 Japan
[2] Dept. of Computer Science and Comm. Eng., Kyushu University, 6–1 Kasuga-koen, Kasuga, Fukuoka 816-8580 Japan
[3] Dept. of Informatics, Kyushu University, 6–1 Kasuga-koen, Kasuga, Fukuoka 816-8580 Japan

## 1 Introduction

Integrating main memory (DRAM) and processors into a single chip, or merged DRAM/logic LSI, makes it possible to exploit high on-chip memory bandwidth by widening on-chip bus and on-chip DRAM array. In addition, from energy point of view, the integration brings a significant improvement by decreasing the number of off-chip accesses.

For merged DRAM/logic LSIs with on-chip cache memory, we can exploit the high bandwidth by means of replacing a whole cache line at a time. This approach tends to increase the cache-line size if we attempt to exploit the attainable high bandwidth. A large cache-line size gives a benefit of prefetching effect if programs have rich spatial locality. Otherwise, however, it will bring the following disadvantages due to poor spatial locality:

1. A number of conflict misses will take place due to frequent evictions.
2. As a result, a lot of energy will be wasted for on-chip DRAM (main memory) due to a number of main memory accesses.
3. Activating the wide on-chip bus and the DRAM array will also dissipate a lot of energy.

Employing set-associative caches is a conventional approach to solving the first and second problems, because it can improve cache-hit rates by reducing conflict misses. However, since increasing the cache associativity increases cache-access time and energy, it might worsen the performance/energy efficiency of memory systems. In addition, we still have the third problem.

In order to solve all the problems without any cache-access time and energy overheads, we have proposed the *variable line-size cache (VLS cache)* architecture for merged DRAM/logic LSIs [3] [4]. The VLS cache exploits the high bandwidth by means of larger cache lines. At the same time, it can alleviate the negative effects of large cache line by partitioning it into multiple small cache lines (sublines). Activating only the DRAM subarrays corresponding to the sublines to be replaced makes a significant energy reduction. In [3] [4], we

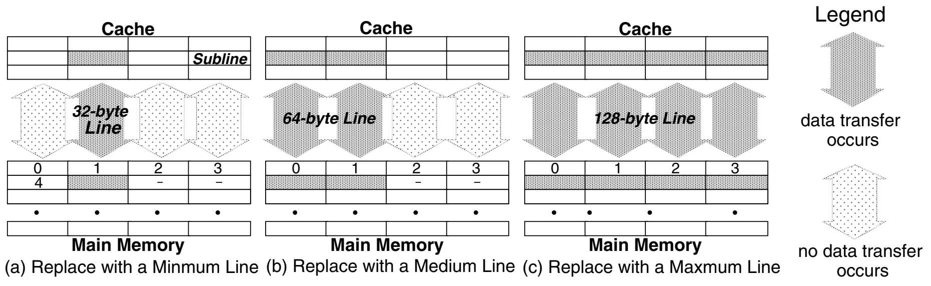


**Fig. 1.** Mechanism of Variable Line-Size Cache

have discussed only the performance attainable in the VLS cache. This paper evaluates both the performance and energy improvements achieved by the VLS cache architecture.

## 2 Variable Line-Size Cache Architectures

### 2.1 Concept

In the VLS cache, an SRAM-cell array (cache) and a DRAM-cell array (main memory) are divided into several subarrays. Data transfer for cache replacements is performed between corresponding SRAM and DRAM subarrays. A block of data associated with a single tag in the cache is referred as *subline. Line* is a block of data transferred between cache and main-memory for replacements.

Fig. 1 shows the mechanism of variable line-size cache. If programs have rich spatial locality, a line consists of many sublines and a large number of sublines would be involved on cache replacements. Contrarily, a few number of sublines would be replaced when programs have poor spatial locality. In case of Fig. 1, the cache-line sizes of 32-byte, 64-byte, and 128-byte are provided. Activating the DRAM subarrays and the on-chip buses corresponding to the replaced sublines reduces the energy consumed for accessing to the on-chip main memory.

The effectiveness of VLS cache depends on how much the cache can choose appropriate line sizes (i.e., the number of sublines to be replaced). There are at least two approaches to the line size optimization: one is a static determination based on prior analysis; the other is a dynamic determination using hardware supports.

### 2.2 Statically Variable Line-Size Cache

The statically variable line-size cache (S-VLS cache) changes its line size program by program [3]. Application programs are analyzed by using cache simulators in advance in order to determine an appropriate line size. In case that the S-VLS cache provides 32-byte, 64-byte, and 128-byte lines, for example, we can

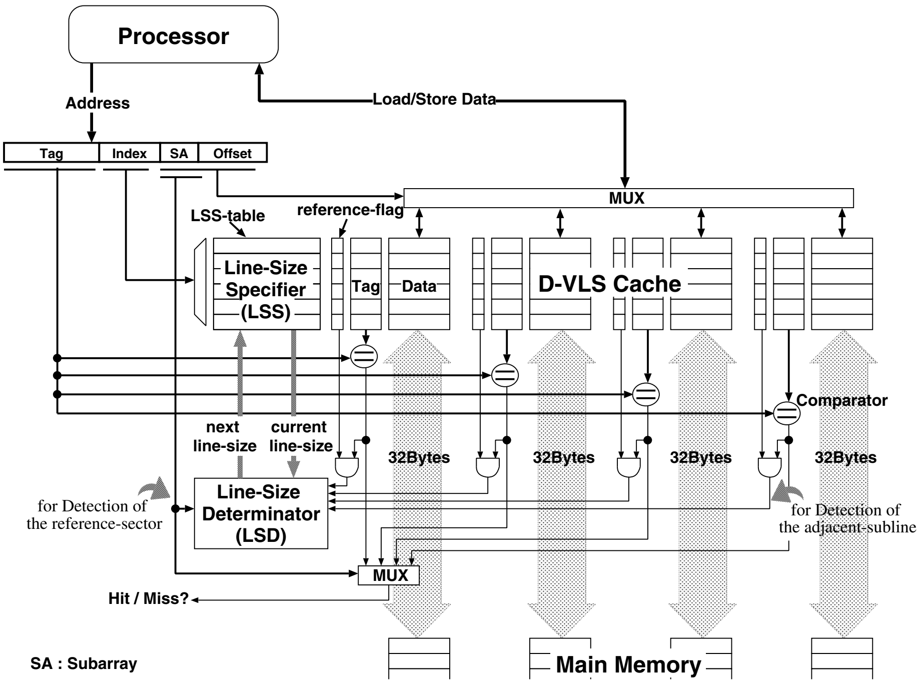


**Fig. 2.** Block Diagram of a Direct-Mapped D-VLS Cache

determine the appropriate line size in the following manner. First, the program is simulated three times to measure cache-hit rates, each of which assumes one of fixed 32-byte, 64-byte, and 128-byte line sizes. Then we choose the best line size as the appropriate line size. The line-size information might be explicitly designated by a control register.

## 2.3 Dynamically Variable Line-Size Cache

It may be possible to adopt the static approach explained in Section 2.2 when target programs have regular access patterns within well-structured loops. However, a number of programs have non-regular access patterns. In addition, the amount of spatial locality may vary both within and among program executions. Against the static approach, the dynamically variable line-size cache (D-VLS cache) selects adequate line sizes based on recently observed data reference behavior at run time. Fig. 2 illustrates the block diagram of a direct-mapped D-VLS cache having three line sizes, 32 bytes, 64 bytes, and 128 bytes. The cache has some hardware components for optimizing the line size. Tag comparison is performed at each subarray. The details of D-VLS cache behavior and an algorithm optimizing the cache-line size have been described in [4].

## 3 Evaluations

We have evaluated the performance/energy efficiency of on-chip memory systems employing the following conventional and VLS caches:

- Fix128 and Fix128W2 : 16 KB conventional caches having fixed 128-byte cache lines. Fix128 is a direct-mapped cache and Fix128W2 is a 2-way set-associative cache.
- Fix128db : 32 KB conventional direct-mapped cache having fixed 128-byte cache lines.
- SVLS128-32 and DVLS128-32 : 16 KB direct-mapped variable line-size caches having 32-byte, 64-byte, and 128-byte cache lines. Line-size optimizations for them are based on the static approach (SVLS) explained in Section 2.2 and the dynamic approach(DVLS) explained in Section 2.3, respectively.

We have measured cache-miss rates and line-replacement counts for each model using benchmark programs: seven integer programs with train input and two floating-point programs with test input from the SPEC95 benchmark suite. Furthermore, to realize more realistic execution on general purpose processors, a benchmark set (*Mix-IntFp*) is used. The programs in the benchmark set are assumed to run in multiprogram manner on a uni-processor system, and a context switch occurs per execution of one million instructions. *Mix-IntFp* is formed by two integer programs and one floating-point program, and three billion instructions are executed. All programs are compiled by GNU CC with the "–O2" option, and are executed on Ultra SPARC workstations. The address traces are captured by QPT [2]. In this evaluation, we have introduced $AMAT$ (Average Memory Access Time) and $AMAE$ (Average Memory Access Energy) as performance and energy metrics of memory systems.

$$AMAT = T_{Cache} + MissRate \times 2 \times T_{MainMemory}. \tag{1}$$

$$AMAE = E_{Cache} + MissRate \times 2 \times E_{MainMemory}. \tag{2}$$

$T_{Cache}$ and $E_{Cache}$ are access time and energy consumption for cache, and $T_{MainMemory}$ and $E_{MainMemory}$ are access time and energy consumption for main memory, respectively. We have assumed that two main-memory accesses are performed for a cache replacement (one for write-back and one for cache refill).

### 3.1 Cache-Access Time and Energy

The structure of direct-mapped VLS caches having 32-byte, 64-byte, and 128-byte lines is similar to that of conventional set-associative caches having 32-byte lines as shown in Fig. 2. In the conventional set-associative cache, the path which chooses a way based on tag-comparison results determines the cache-access time. However, that critical path does not appear in the VLS caches because the target

**Table 1.** Cache-Access Time and Energy

| Cache | $T_{Cache}$ $[Tunit]$ | $E_{Cache}$ $[Eunit]$ |
|---|---|---|
| Fix128 | 1.000 | 1.000 |
| Fix128W2 | 1.470 | 1.160 |
| Fix128db | 1.195 | 1.838 |
| SVLS128-32 | 1.000 | 1.051 |
| DVLS128-32 | 1.000 | 1.090 |

subarray can be chosen regardless of tag-comparison results. Namely, control signals for the subarray selection are made from the reference address directly. Therefore, we assume that the cache-access time of direct-mapped VLS caches (SVLS128-32, DVLS128-32) is the same as that of conventional direct-mapped cache having same cache size and same associativity (Fix128).

Increasing cache associativity consumes more energy because it increases the total number of bit-lines, precharging circuits, sense amplifiers, and so on. Similarly, increasing the cache size dissipates a lot of energy due to the increase in bit-line capacitances. On the other hand, the VLS caches do not have this kind of energy overheads, because the cache size and associativity of Fix128 are maintained. The VLS caches need to dissipate the energy for extra tag comparisons; a tag comparison is performed at each subarray. However, the total number of bit-lines to be activated for a tag-memory access is much smaller than that for a data-memory (cache line) access. Therefore, the energy overhead for the extra tag comparison is small. In addition, although the D-VLS cache needs to read a 2-bit LSS (line-size specifier) and four 1-bit reference flags for run-time line-size optimization, this energy overhead is also trivial.

To find the cache-access time of each model, we have used the CACTI 2.0 which is the updated version of CACTI model [5] [8]. In addition, we have calculated the cache-access energy for each organization based on Kamble's model [6]. We referred the load capacitance of each node defined in [7], which is based on the 0.8 micron CMOS cache design described in [8]. In this evaluation, we refer to the cache-access time and cache-access energy of Fix128 as $T_{unit}$ and $E_{unit}$, respectively. Table 1 summarizes calculation results.

## 3.2 Cache-Miss Rate

Table 2 shows cache-miss rates on conventional and VLS caches. For some programs, the VLS caches (SVLS128-32, DVLS128-32) can achieve almost all the same or lower cache-miss rates than the double-size conventional direct-mapped cache (Fix128db). However, increasing associativity produces much better results for many programs. In average, conventional approaches to improving cache performance, increasing cache size (Fix128db) or cache associativity (Fix128W2), achieve lower cache-miss rates than the VLS caches.

**Table 2.** Cache-Miss Rates

| Program | Fix128 | Fix128W2 | Fix128db | SVLS128-32 | DVLS128-32 |
|---|---|---|---|---|---|
| 099.go | 0.1024 | 0.0695 | 0.0541 | 0.0571 | 0.0638 |
| 124.m88ksim | 0.0202 | 0.0045 | 0.0068 | 0.0167 | 0.0153 |
| 126.gcc | 0.0611 | 0.0344 | 0.0349 | 0.0535 | 0.0526 |
| 130.li | 0.0341 | 0.0203 | 0.0226 | 0.0341 | 0.0358 |
| 132.ijpeg | 0.0244 | 0.0048 | 0.0068 | 0.0195 | 0.0175 |
| 134.perl | 0.0542 | 0.0230 | 0.0295 | 0.0332 | 0.0286 |
| 147.vortex | 0.0505 | 0.0292 | 0.0307 | 0.0361 | 0.0374 |
| 101.tomcatv | 0.0633 | 0.0182 | 0.0546 | 0.0633 | 0.0578 |
| 104.hydro2d | 0.0481 | 0.0217 | 0.0259 | 0.0481 | 0.0295 |
| Mix-IntFp | 0.0597 | 0.0327 | 0.0311 | 0.0452 | 0.0377 |

## 3.3 Main-Memory-Access Time and Energy

The main-memory-access time ($T_{MainMemory}$) and energy ($E_{MainMemory}$) depend on the memory size, organization, process technology, and so on. In this evaluation, we assume that the main-memory-access time including the delay for data transfer between the cache and the main memory is ten times longer than the access time of the 16 KB direct-mapped conventional cache having 128-byte lines (i.e., $T_{MainMemory} = 10 \times Tunit$).

For the main-memory-access energy ($E_{MainMemory}$), we assume that there is no energy dissipation for DRAM refresh operations in order to simplify the evaluation. Thus, for the on-chip memory-path architectures with a conventional cache, energy dissipated for main-memory accesses (i.e., $MissRate \times 2 \times E_{MainMemory}$) depends only on the total number of main-memory accesses. In other words, only cache-miss rates affect the main-memory-access energy. On the other hand, since the VLS caches activate only the DRAM subarrays corresponding to sublines to be replaced, the energy consumed for accessing to the on-chip main memory depends not only on cache-miss rates but also on line sizes (i.e., the number of sublines to be involved in cache replacements). Accordingly, the main-memory-access energy ($E_{MainMemory}$) in Equation (2) can be expressed as follow:

$$E_{MainMemory} = 10 \times Eunit \times \frac{AverageLineSize}{128bytes}. \quad (3)$$

Here, we assume that the energy dissipated for an access to the 128-byte width on-chip DRAM is ten times larger than the cache-access energy of Fix128 [1]. The right factor ($\frac{AverageLineSize}{128bytes}$) in Equation (3) denotes a coefficient which represents the average number of 32-byte DRAM subarrays activated per cache replacement. The average line size of conventional caches is 128 bytes, so that the value of the coefficient is 1. While that of VLS caches depends on the characteristics of programs. For instance, if the average line size is 64 bytes, the value of the coefficient is 0.5.

**Table 3.** Average Line Size and Replace Count on VLS caches

| Program | S-VLS | D-VLS | | | |
|---|---|---|---|---|---|
| | Ave. Line Size [B] | Replace Count | | | Ave. Line Size [B] |
| | | 32 [B] | 64 [B] | 128 [B] | |
| 099.go | **32.00** | 6,445,160 | 1,724,674 | 389,746 | **42.82** |
| 124.m88ksim | **64.00** | 317,746 | 53,858 | 68,353 | **50.83** |
| 126.gcc | **64.00** | 10,092,540 | 3,463,487 | 1,468,861 | **48.76** |
| 130.li | **128.00** | 1,190,072 | 426,488 | 189,570 | **49.63** |
| 132.ijpeg | **64.00** | 3,530,649 | 1,179,064 | 1,246,695 | **58.43** |
| 134.perl | **32.00** | 7,987,886 | 5,250,134 | 3,849,457 | **63.46** |
| 147.vortex | **32.00** | 19,805,372 | 3,593,130 | 1,416,595 | **42.11** |
| 101.tomcatv | **128.00** | 23,539,313 | 2,608,352 | 2,650,269 | **43.73** |
| 104.hydro2d | **128.00** | 3,784,227 | 860,802 | 6,175,600 | **89.34** |
| Mix-IntFp | **82.60** | 17,005,515 | 4,564,846 | 7,577,526 | **61.97** |

Table 3 shows the average line size on the S-VLS cache (SVLS128-32) and the D-VLS cache (DVLS128-32) for each program. The table also reports the breakdown of cache-replacement count for each line size in the D-VLS cache, and the minimum and maximum average cache-line sizes are 42.82 bytes for *099.go* and 89.34 bytes for *104.hydro2d*, respectively. For all programs, average line size of the S-VLS cache is 75 bytes, and that of D-VLS cache is 55 bytes.

Fig. 3 depicts the energy consumed for accessing to the on-chip main memory ($CMR \times 2 \times E_{MainMemory}$). All results are normalized to the memory system employing the 16 KB conventional direct-mapped cache having 128-byte lines (Fix128). Although the cache-miss rates of the VLS caches are higher than those of the conventional caches (Fix128W2, Fix128db), the VLS caches make

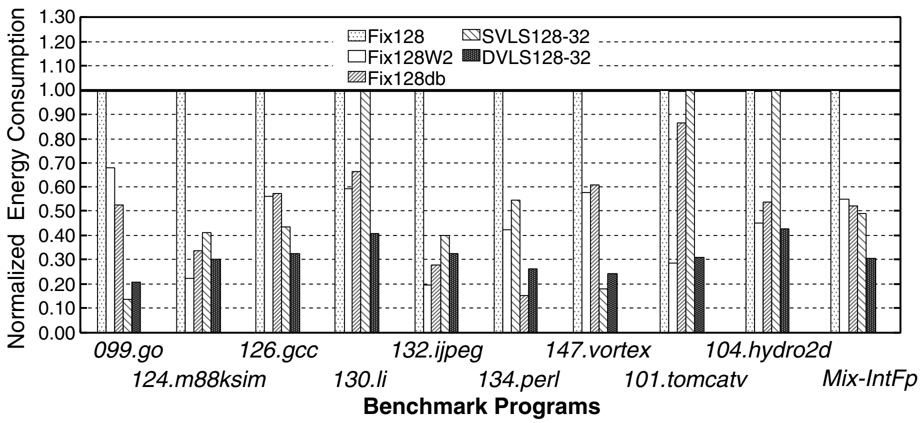


**Fig. 3.** Main-Memory Access Energy ($CMR \times 2 \times E_{MainMemory}$)

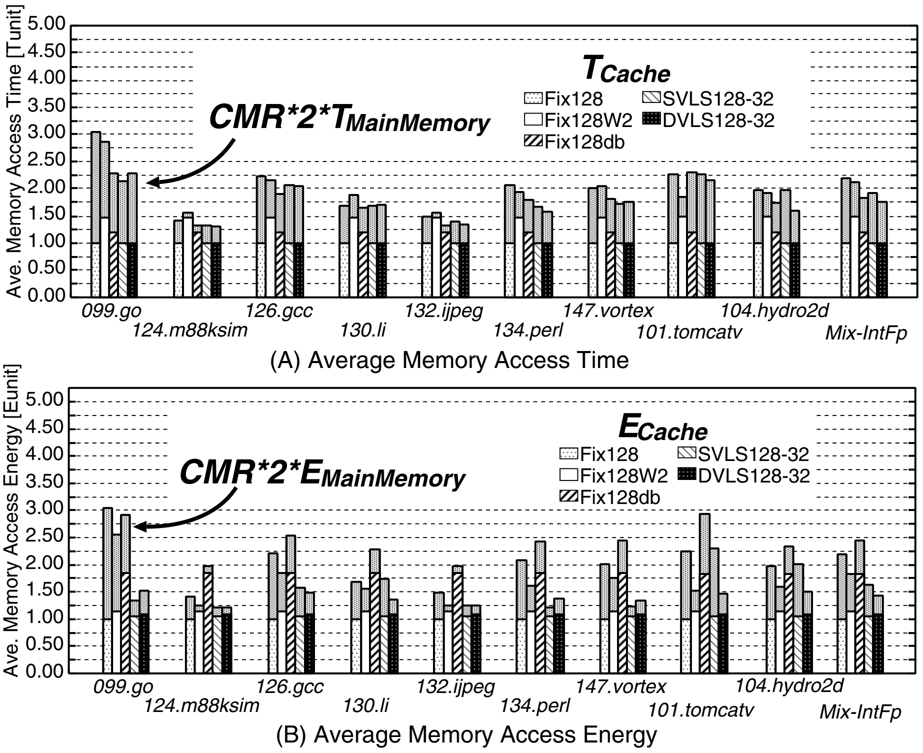


**Fig. 4.** Average Memory Access Time and Energy

significant advantages of energy reduction by reducing both the total number of main-memory accesses and that of DRAM subarrays activated. For *Mix-IntFp*, the D-VLS cache (DVLS128-32) produces about 70 % main-memory energy reduction from the base organization (Fix128), which is about 20 % better than the low-miss-rate conventional caches (Fix128W2, Fix128db).

## 3.4 Performance/Energy Efficiency

Fig. 4 (A) shows the performance ($AMAT$) of each memory system in case that the access time of on-chip DRAM is ten times longer than $T_{unit}$. Increasing associativity improves cache-miss rates. However, Fix128W2 does not produce good result for some programs due to the cache-access-time overhead. On the other hand, the VLS caches can maintain the fast access of direct mapping. Thus, in all programs except for *101.tomcatv*, the VLS caches make significant performance improvements, which are comparable with the doubled size conventional direct-mapped cache (Fix128db).

Fig. 4 (B) shows energy consumption ($AMAE$) of each memory system in case that the energy consumed for accessing to on-chip DRAM is ten times

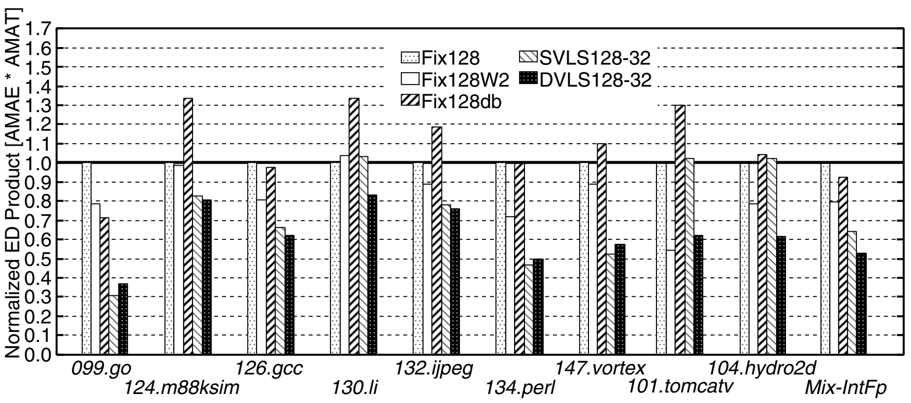


**Fig. 5.** Energy Delay Products ($AMAT \times AMAE$)

larger than $E_{unit}$. The conventional approaches to improving cache performance (Fix128W2, Fix128db) reduce cache-miss rates, so that the main-memory-access energy is reduced from the base organization (Fix128). However, Fix128db degrades the total energy efficiency because the cache-access energy is large. In conventional caches, Fix128W2 produces the best results for all programs. On the other hand, the VLS caches can make energy reductions for main-memory accesses by improving cache-hit rates and by activating on-chip DRAM subararys selectively without large cache-access-energy overhead. Thus, the VLS caches make significant energy reductions. In particular, the D-VLS cache (DVLS128-32) achieves lower energy for all programs than Fix128W2 which is the best organization of conventional approaches.

Finally, in Fig. 5, we show the energy-delay product to evaluate the performance and energy at the same time. For each program, all results are normalized to Fix128. In conventional caches, the performance improvement achieved by increasing cache size (Fix128db) is negated by the large energy dissipation. Contrarily, energy improvement produced by increasing associativity (Fix128W2) is negated by low-performance due to long cache-access time. The VLS caches do not have this kind of negations because they can produce both the performance and energy improvements. In the best case of *099.go*, the VLS caches reduce the energy-delay product more than 60 % from the conventional direct-mapped cache (Fix128). While the reductions produced by conventional approaches, increasing cache associativity or cache size, are from 20 % to 30 %.

## 4 Conclusions

In this paper, we have described the variable line-size cache (VLS cache), which is a novel cache architecture suitable for merged DRAM/logic LSIs. To evaluate the performance/energy efficiency of the VLS caches, we have simulated many benchmark programs on the VLS caches and on conventional caches. In the simulation results for *Mix-IntFp* benchmark set which includes two integer programs and one floating-point program, it is observed that a statically VLS cache and a dynamically VLS cache reduce the energy-delay product by 35 % and 47 %, respectively, compared to a conventional cache having the same cache size and the same associativity.

Employing merged DRAM/logic LSIs is one of the most important approaches for future computer systems, because it can achieve high-performance and low-power at the same time by eliminating the chip boundary between the processor and main memory. We can obtain more improvement of performance/energy efficiency by exploiting the attainable high on-chip memory bandwidth effectively. The VLS cache architecture is applicable to any merged DRAM/logic LSIs. In particular, the dynamically VLS cache is more promising. Because it does not require any modification of instruction set architectures, the full compatibility of existing object codes can be kept.

## References

1. Fromm, R., Perissakis, S., Cardwell, N., Kozyrakis, C., McGaughy, B., Patterson, D., Anderson, T., and Yelick, K., "The Energy Efficiency of IRAM Architectures," *Proc. of the 24rd Annual International Symposium on Computer Architecture*, pp.327–337, May 1997.
2. Hill, M. D., Larus, J. R., Lebeck, A. R., Talluri, M., and Wood, D. A., "WARTS: Wisconsin Architectural Research Tool Set," *http://www.cs.wisc.edu/~larus/warts.html*, University of Wisconsin - Madison.
3. Inoue, K., Koji, K., and Murakami, K., "High Bandwidth, Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs," *IEICE Transactions on Electronics*, Vol.E81-C, No.9, pp.1438–1447, Sep. 1998.
4. Inoue, K., Kai, K., and Murakami, K., "Dynamically Variable Line-Size Cache Exploiting High On-Chip Memory Bandwidth of Merged DRAM/Logic LSIs," *Porc. of the 5th International Symposium on High-Performance Computer Architecture*, pp.218–222, Jan. 1999.
5. Jouppi, P. N., *http://www.research.digital.com/wrl/people/jouppi/CACTI.html*
6. Kamble, M. B., and Ghose, K., "Analytical Energy Dissipation Models For Low Power Caches," *Proc. of the 1997 International Symposium on Low Power Electronics and Design*, pp.143–148, Aug. 1997.
7. Kamble, M. B., and Ghose, K., "Energy-Efficiency of VLSI Caches: A Comparative Study," *Proc. of the 10th International Conference on VLSI Design*, pp.216–267, Jan. 1997.
8. Wilton, S. J. E., and Jouppi, N. P., "An Enhanced Access and Cycle Time Model for On-Chip Caches," *Digital WRL Research Report 93/5*, July 1994.

# The DIVA Emulator: Accelerating Architecture Studies for PIM-Based Systems

Jeff La Coss[1]

University of Southern California
Information Sciences Institute

## 1 Introduction

*DIVA* (*Data IntensiVe Architecture*), employs Embedded DRAM (EDRAM) technology [Iyer99] to overcome memory bandwidth limitations by tightly coupling a single processor to a large on-chip storage array to produce a device capable of dual roles as system "smart" and "dumb" memory. Communication between "nodes" (processor-memory pairs) occurs on a special chip-to-chip interconnect, off-loading the system memory bus. Coarse-grain parallelism may be further extended by implementing multiple processor-memory "nodes" per PIM chip. The DIVA system will employ PIMs in a workstation "smart memory" system capable of large amounts of processing.

The DIVA emulator is a programmable logic resource for accelerating detailed evaluation of PIM architecture. Based on commercial FPGAs, the emulated logic model directly executes code at least 50 times faster, and at much finer granularity, than our simulator. As a result. the emulator provides a platform for early development of system software and evaluation of algorithms that exploit the fine-grain parallelism available in DIVA PIMs.

### 1.1 Overview of DIVA System Architecture

Figure 1 shows the major control and data connections within a processor-memory node. Information flows into and out of the node via the communication port or the host memory port. The heart of the DIVA node is a wide data bus used for all transfers between major functional units. Arbitration between the host and the node adds insignificant delay to the host memory access time when the PIMs are used as conventional memory.

Node processing logic consists of an execution control pipeline to coordinate the activity of a standard 32-bit integer scalar datapath (registers plus ALU), a 64-bit floating point datapath, and a special 256-bit WideWord datapath that processes data at the

1. *Author's address*: University of Southern California Information Sciences Institute. 4676 Admiralty Way, Suite 1001, Marina Del Rey CA 90292 USA. *email* jlacoss@isi.edu, *WWW:* http://www.isi.edu/~jlacoss

memory bandwidth. Register-register moves between datapaths are supported for efficient interaction between different types of processing. The execution control pipeline, scalar datapath, and floating point datapath may be viewed as a conventional microprocessor and can be programmed as such, enabling evolutionary software development. Users may also exploit coarse-grain parallelism offered by the PIMs by simply programming multiple nodes in a conventional sense. However, users may also exploit fine-grain parallelism by using the WideWord datapath. Further DIVA architectural details can be found in [Hall99].

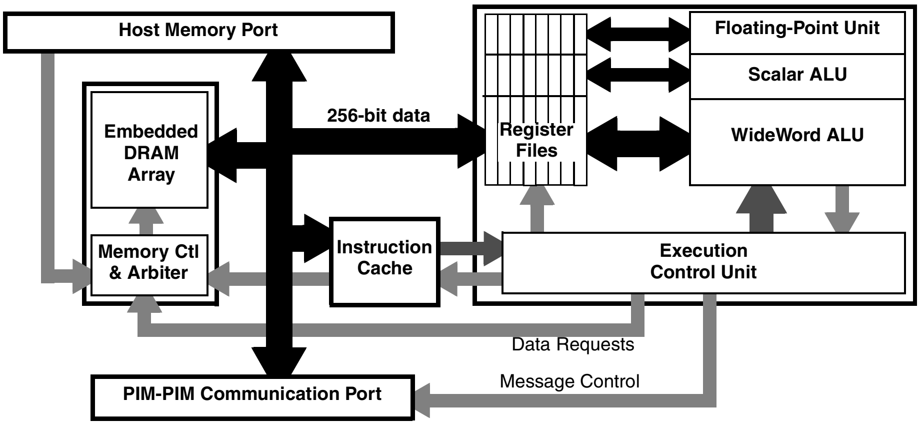


**Fig. 1:** DIVA PIM Node Organization

## 1.2 Simulation

Several forms of software simulation are well known. *High-level* simulators mimic system behavior, but deliver information at the input/output level only. *Register-transfer-level* simulators maintain state information for the functional modules used in a system. *Gate-level* simulators use very fine-grain models to describe the system at the lowest level of detail - the logic gates used to implement major units. This 1:1 mapping between model and target, or *fidelity*, minimizes the chance of error during physical system design and implementation. Unfortunately, the computing power needed to execute such simulations and manage required information stresses the capabilities of advanced workstations, resulting in very long simulation times. DSIM, the DIVA simulator derived from RSIM, executes approximately twenty-five thousand instructions per second on a SUN Ultra20 workstation. The need for efficient system verification justifies the cost of developing of specialized hardware emulators.

## 1.3 Emulation

Emulation models systems in hardware quite different from the target implementation. Emulators based on reprogrammable logic devices allow new logic configurations to be assessed on a single platform. *Reconfigurable application-specific computing* can be considered an extreme case of emulation: features of a computation - hardware and software - not required by a given application are optimized away during the compilation process. The most accurate type of emulation is *gate-level*, which provides a hardware model for each fine-grain logic element to rapidly execute the desired functions of the anticipated system. There are drawbacks to this high-fidelity approach, however, as very large amounts of reconfigurable hardware resources can be required. EETimes published an account where the first Intel Pentium was emulated with a reconfigurable platform that required 100 square feet of floor space.

## 2 The DIVA Emulator

The DIVA emulator mimics the behavior of the DIVA VLSI PIM node at a moderate clock rate (up to ten million instructions per second). Some tradeoffs are required because the Xilinx "Virtex" FPGA devices do not provide a substrate of "gates" for constructing logic, meaning the emulation must sacrifice some fine-grain fidelity. Originally intended to model four nodes, the current version supports a maximum of two because the WideWord (256-bit) datapath logic consumes FPGA resources, requiring the node processor to be partitioned across multiple devices. However, *per-clock fidelity* is maintained, so that at each user-visible time marker all state information available to the debugging support environment is indistinguishable from that of the anticipated VLSI device. This fidelity ensures that program code compiled for the VLSI PIM will run and produce identical results on the emulator hardware, giving architects a platform for rapidly evaluating design trade-offs.

### 2.1 Physical Implementation

The DIVA emulator is implemented on a full-size PCI card and mezzanine board. These boards contain all logic required for system bus interface, power conversion and monitoring circuits, as well as FPGAs with dedicated DRAM and SRAM to support emulation of complex circuits of up to several million gates.

Circuit topology is constrained by printed-circuit interconnect, physical packaging of the commercial devices, and pin limitations of the FPGAs. DRAM is implemented as 64 bit words per FPGA, requiring WideWord memory accesses to be emulated by quadword read and write operations. These are controlled by "microstates," invisible to the user, within the instruction clock cycle. Microstates also define the protocol used to pass data between the FPGAs of the emulator: at different points in time, instructions, scalar (32-bit) data, and WideWord ALU condition codes are transmitted over the limited interconnect available for this communication.

As shown in Figure 2, the emulator board has ports for the PIM-to-PIM communication channels, enabling emulation of PIM arrays. The communication logic is emulated separately from node logic. The narrow paths between FPGAs also require use of microstates to mimic single-clock PIM communication events. The emulator also has a port for a host memory bus interface. The goal of this feature is to use the emulator *in situ* with the host system, operating as a very slow DRAM. The implementation of these interfaces is pending.

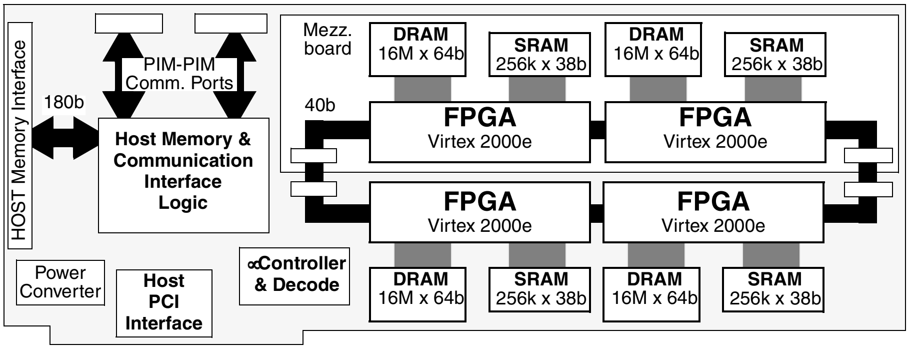


**Figure 2: DIVA Emulator Implementation**

## 2.2 Operation

Users develop a logic description in VHDL, compile it, and load the resulting configuration file into the emulators FPGAs via a Linux user interface. This software also provides the user with direct fine-grain control and monitoring of the logic embedded in the FPGAs. A well-understood DSP application has been demonstrated and is used as a benchmark to evaluate changes to the user interface as well as determine emulator performance.

## 3 Summary and Conclusion

This paper describes the DIVA PIM hardware emulator, designed to evaluate architectural features of a new PIM smart-memory coprocessor for a conventional host system. We briefly describe the target architecture, the emulator platform and system support software, and an example of applications run on the emulated logic. With the emulator running at an extreme worst-case speed of one million instructions per second, we achieve execution speedup factors over fifty at a level of detail unapproachable by software simulation techniques.

The DIVA emulator is a work in progress, with new logic functions being added on a regular basis. The evaluation environment runs several large-scale application exploiting processing with the wide datapaths. In the future, we also plan to implement IEEE 32- and 64-bit floating point operation.

## 4 References

[Hall99] *Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture*. M. Hall, *et al.* Supercomputing 99.

[Iyer99] *Embedded DRAM technology: opportunities and challenges*. Iyer, Subramanian and Howard Kalter, IEEE Spectrum Apr. 1999.

[RSIM] http://www-ece.rice.edu/~RSIM

# Compiler–Directed Cache Line Size Adaptivity [⋆]

Dan Nicolaescu[1], Xiaomei Ji[1], Alexander Veidenbaum[1], Alexandru Nicolau[1], and Rajesh Gupta[1]

Department of Information and Computer Science
444 Computer Science, Building 302
University of California Irvine
Irvine, CA 92697–3425
{dann,xji,alexv,nicolau,rgupta}@ics.uci.edu

**Abstract.** The performance of a computer system is highly dependent on the performance of the cache memory system. The traditional cache memory system has an organization with a line size that is fixed at design time. Miss rates for different applications can be improved if the line size could be adjusted dynamically at run time. We propose a system where the compiler can set the cache line size for different portions of the program and we show that the miss rate is greatly reduced as a result of this dynamic resizing.

## 1 Introduction

The area available for on–chip caches is limited and the size and associativity of a cache for a given processor cannot be significantly increased without causing an increase in the cycle time. Currently in a given technology implementation processor designers decide the size of cache lines by considering different tradeoffs between speed and latency. But this tradeoff also influences the miss rate of the cache system. We show that this decision has a great impact on the miss rate of the memory system.

We propose a simple system that allows the cache line size to vary at run time. To achieve this we augment the ISA with a single extra instruction that sets the line size. A compiler can insert this instruction in the code at points it determines suitable by either static code analysis or profile–directed feedback.

While the hardware modifications are modest, the following questions need to be answered to determine the feasibility of the approach:

1. When should the cache line size be changed,
2. How often is it necessary to reconfigure,
3. What is the optimal reconfiguration policy?

On one hand it would not be feasible to change the cache line size every few instructions as the overhead associated with such reconfiguration would make

[⋆] This work was supported in part by the DARPA ITO under Grant DABT63-98-C-0045.

the approach prohibitively expensive. On the other hand if we reconfigure too infrequently, e.g. once per function call, we might miss some optimization opportunities because a function may contain a number of loops, each of them with a distinct cache behavior.

It has been shown that the majority of dynamic instructions in a program are executed in innermost loops. An inner loop is also likely to have reasonably stable spatial/temporal locality characteristics. This suggests that an inner loop may be a good place to change the cache line size and maintain the setting for the duration of such a loop. In this paper we study the performance of changing the cache line size at loop level and show that such an approach is feasible.

We currently use a profile–based mechanism for the control of adaptation by the compiler. Future work will study the opportunity to use compile–time analysis for making adaptivity decisions.

## 2 System Organization

The system being studied consists of a 3–level memory hierarchy. The cache line size is reconfigurable at run time using a special instruction. The set of sizes is: 8, 16, 32, 64, 128 and 256 bytes. A fully associative write buffer is also used.

The L1 cache is direct mapped and the hit latency is assumed to be 1 cycle. The L1 bus transfer takes 2 cycles. L2 is a 2–way set–associative with the access latency of 15 cycles. The main memory access latency is 100 cycles.

## 3 Experimental Infrastructure

### 3.1 Simulator

The framework provided by the ABSS [2] simulation system is used in this study. ABSS is a simulator that runs on SUN Sparc systems and is derived from the MINT simulator [3].

The ABSS simulator consists of 5 parts: augmentor, thread management, cycle-counting libraries, user-defined simulator of the memory system and the application program.

The augmentor program (called *doctor*) parses the original application assembly code, and adds instrumentation code that sends information about the loads and stores executed by the program to the simulator.

Our custom memory architecture simulator simulates a 3–level memory hierarchy, the cache line for the L1 cache is changeable at run time via commands embedded in the simulated program.

### 3.2 Compilation

We have used version 2.95 of the GCC compiler collection to conduct all the experiments. The compiler back–end was modified to emit special code sequences before entering a loop, or on the code path for exiting a loop. Given that the compiler back–end is common to the C and Fortran77 compiler we were able to use this instrumentation for compiling all the SPEC95 benchmarks.

The code sequences were used for adjusting the cache line size, and for collecting statistics and identifying the loop (source file name and line number), and signaling to the cache simulator that a loop is being entered or exited.

All the benchmarks where compiled using the -O2 optimization flag, the target instruction set was SPARC V8plus.

## 4 Experiments

We have run the simulations for a memory system using different cache line sizes.

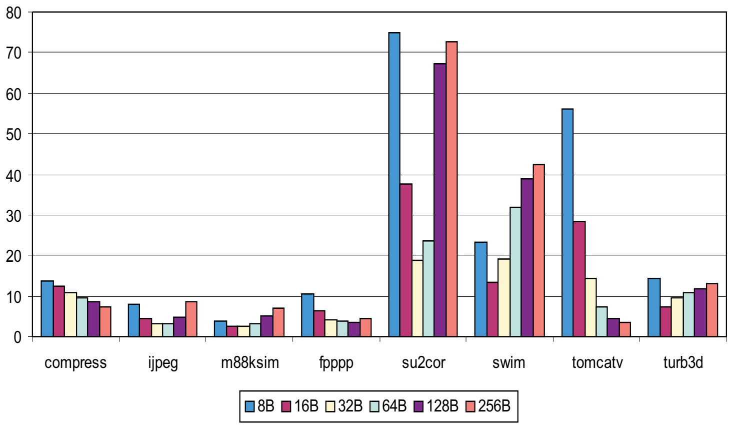


**Fig. 1.** L1 miss rate for the loop containing the most memory references.

The results shown in Figure 1 are miss rate for the loop that contains the most memory references for each benchmark is shown. It can be observed that no individual cache line size gets the minimum miss rate for all benchmarks, and that there is no rule for all benchmarks that could determine the optimal cache line size.

Figure 2 shows that even for the same benchmark, different loops have better miss rates for different cache line sizes. Based on these two fact we can conjecture that adapting the line size on loop boundaries improves the miss rate.

We used profiling to determine the best cache line size for each loop, we run the benchmarks using the training input set, determined for each loop what is the cache line size that generates the minimum miss rate and used that data to run the benchmarks using a compiler generated instruction to change the cache line size to the one that was determined to generate the minimum number of misses. This approach is practical since we have determined that the data

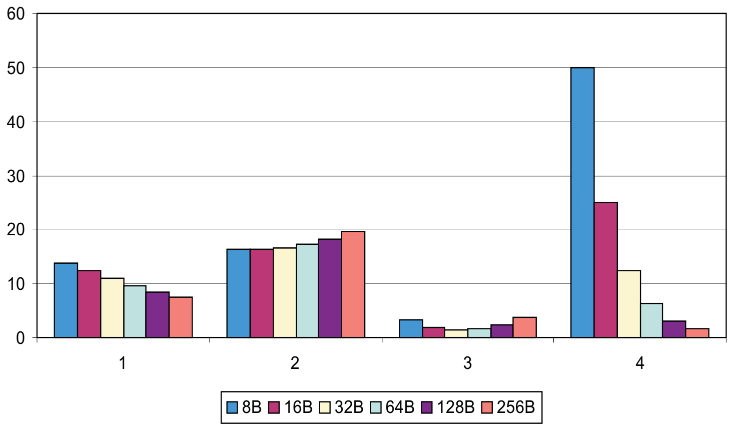


**Fig. 2.** L1 miss rate for the loops that generate most memory references for the compress benchmark

obtained using the small, manageable training input sets extrapolates well to the actual application. The results are shown in Figure 3. It can be seen that the miss rate always improves, sometimes by a wide margin.

Another interesting observation can be made from Figure 3. The "worst case" data is obtained by using the profile data for setting the line size in such way to maximize the miss rate. It can be observed that this worst case is in all cases very close to the miss rate for at least one of fixed line sizes. So it is likely that for about any fixed line size there will exist an application that will have a very high miss rate, therefore cache line size adaptability is a worthwhile feature for a general purpose processor that has to run well a variety of applications.

## 5 Conclusions and Future Work

We have shown that adapting the cache line size on a per loop basis improves the cache miss rate. We have used a profile base approach, future work will determine the cache line size at compile time using analytical approaches and we are also working on using hardware based approaches to dynamically change the cache line size.

## References

1. Teresa L. Johnson and Wen mei Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
2. D. Sunada, D. Glasco, and M. Flynn. ABSS v2.0: SPARC simulator. Technical Report CSL-TR-98-755, Stanford University, 1998.

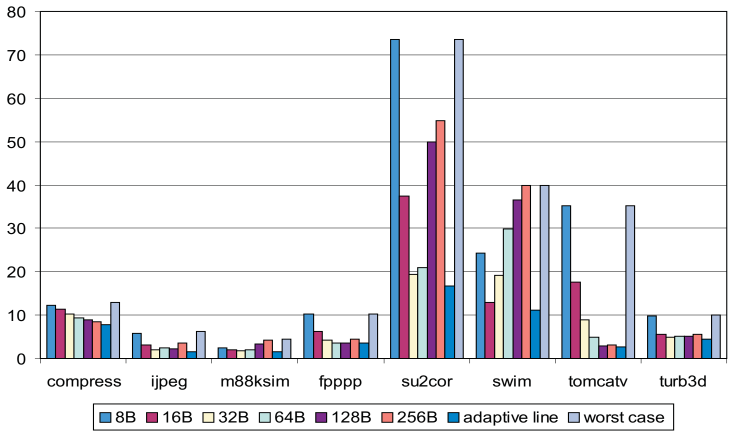


**Fig. 3.** L1 miss rate for different cache line sizes, and for an adaptive cache

3. Jack E. Veenstra and Robert J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Intl. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, 1994.
4. Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. Adapting cache line size to application behavior. In *Proceedings ICS'99*, June 1999.
5. Peter Van Vleet, Eric Anderson, Lindsay Brown, Jean-Loup Baer, and Anna Karlin. Pursuing the performance potential of dynamic cache line sizes. In *Proceedings of 1999 International Conference on Computer Design*, November 1999.

# Summary of Question/Answer Sessions for Workshop Presentations

Workshop Notes

These notes summarize the question and answer sessions held after each presentation. They are a combined collection of notes from Mark Oskin and Frederic T. Chong.

## 1 Memory Technology

### Embedded DRAM: Technology and Challenges

*Subramanian S. Iyer, IBM Microelectronics*

The DRAM process lags about two generations in performance and one generation in density over logic processes. The IBM embedded-DRAM (eDRAM) cell is 1.3X the size of conventional DRAM but the overall area of conventional DRAM is greater due to lower drive. Consequently, as a rough approximation it is the same size. DRAM storage devices are not scaled with process technology. Using the IBM eDRAM process costs about 20-25% more than a conventional DRAM process. There is a lower device yield due to MDL. Certain areas, such as the network processor market may benefit from a better cost-performance target, even though the integration cost of using eDRAM is increased. The access time to on-chip eDRAM is about 4.2ns, which is approaching the access time of SRAM. Finally, while there is some variety in the DRAM macro blocks, most customers use only one or two types.

### A 64MBit Mesochronous Hybrid Wave Pipelined Multibank DRAM Macro

*Junji Ogawa and Mark Horowitz, Stanford University*

Using the 0.18um commodity process, a 1-2MBit size DRAM macro block is optimal. The area overhead of the hybrid scheme is only about 2%.

### Software controlled Reconfigurable On-chip Memory for High Performance Computing

*Hiroshi Nakamura, Masaaki Kondo, and Taisuke Boku, University of Tokyo, and the University of Tsukuba.*

The Quantum Chromo-Dynamics (QCD) application assumed 1MB of on-chip memory. The CG application from the NAS parallel benchmarks was not a major contributor to the overall performance. The configuration of the on-chip memory is not altered during application execution due to the high reconfiguration cost.

## 2 Processor and Memory Architecture

**Content-based Prefetching: Initial Results**
*Robert Cooksey, Dennis Colarelli and Dirk Grunwald, University of Colorado*

Coverage is “low” on synthetic, and even lower for real data. It is an open issue on whether this is a problem or not. On the Olden benchmarks, the stride prefetcher worked best. John Carter points out that quad tree prefetching performs well, but you have to try and avoid wasting bandwidth.

**Memory System support for Dynamic Cache Line Assembly** *Lixin Zhang, Venkata K. Pingali, Bharat Chandramouli and John B. Carter, University of Utah*

The pages are not pinned. The mapping can be altered at runtime. The SDRAM timing was extensively modeled in detail.

**Adaptively Mapping Code in an Intelligent Memory Architecture**
*Yan Solihin, Jaejin Lee, and Joseph Torrellas, University of Illinois at Urbana-Champaign, Michigan State University and Los Alamos National Laboratory*

The “ideal” configuration is ideal for the P-MEM system or P-PROC, but not for a system that dynamically switches between the two.

## 3 Applications and Operating Systems

**Blue Gene**
*Marc Snir, IBM*

The target clock rate for the on-chip logic was 500mhz. While running, the machine has 18 million threads, 46,000 chips, and consumes 2MW of power. A single processor contains 512K of DRAM and is about 20 square millimeters in size. Communication between processors is done via polling. The code size is in the 10k range with only selective porting of the runtime environment. Code is replicated on every chip. The data access bandwidth per-chip is on the order of 100 Gbytes. It was found that the 3D mesh configuration has sufficient bandwidth for the applications, and communication becomes latency bound.

### The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems

*Richard C. Murphy, Peter M. Kogge and Arun Rodrigues, University of Notre Dame*

Empty bits are used to support writable objects. The application thread is responsible for returning the PIM and/or flushing data back. Thrashing between threads was not found to be a problem – the address window experiments demonstrate this. The thread stack is relatively small (a few words). Currently, data is migrated by pulling data over once a thread has been moved. Current work is looking at taking a small cache along with the thread to further improve performance. Past research has nottried to increase the number of registers to counteract the data migration issues.

### Memory Management in a PIM-based Architecture

*Mary Hall and Craig Steele, USC Information Sciences Institute*

Multi-tasking is supported by "space sharing". Allocations are done on PIM level blocks. The global address space on the PIMs is specific to one particular process on the host. Current work is adapting Linux to be the host operating system. Operating system support executing in the PIM is minimal: buffer management, network management and thread scheduling is done on chip without host processor intervention. Large-scale allocation of memory is coordinated with the host OS. A small scale OS is on-chip for management of the heap, stack, etc. Predefined global segments is handled by the PIM node miniature OS independently.

## 4 Compiler Technology

### Exploiting On-chip Memory Bandwidth in the VIRAM Compiler

*David Judd, Katherine Yelick, Christoforos Kozyrakis, David Martin, and David A. Patterson, University of California at Berkeley*

There are no issues with data alignment within a bank, unlike MMX instructions. A benefit to virtualizing the hardware is that one does not have to change the compiler every time they change the hardware. It is speculated that a different approach would be to make the compiler parameter driven due to the regularity of the architecture. In VIRAM the software specifies the maximum available parallelism that is found, and then leaves it up to the hardware to use that information. The limitation on the number of address registers is arbitrary, but was chosen because such devices take up a lot of area. The compiler maintains memory consistency. Usually the vector unit is running behind the scalar unit because the vector unit is deeply pipelined. Where there is interaction between the vector and scalar units a memory barrier must be implemented. The most expensive is the scalar after vector memory barrier. Conditional bits, as they are used in VLIW processors, don't come much into play with the multimedia applications studied thus far.

**FlexCache: A Framework for Flexible Compiler Generated Data Caching**
*Csaba Andras Moritz, Matthew Frank, and Saman Amarasinghe, University of Massachusetts, and Massachusetts Institute of Technology*

Without FlexCache ISA support, the instruction stream is polluted, with 4 extra instructions for every single memory instruction. With ISA extension there is no affect on the instruction cache. Currently, results assume all loads go to the software cache. Future work will explore mixing a conventional and a FlexCache. The TLB mechanisms are not implemented. Alias analysis for pointers is very precise, and current work has not looked at relaxing this. The model implements a check so even if incorrect alias analysis happens the model still works.

## 5 Open microphone

Konrad Lai (*Intel*)
Processors are quickly reaching the limits of power density. Memory has lower power density than computational logic. Memory requirements are logarithmic over years. On-chip cache will dominate chip area of future microprocessors – but is there a better way? Even though the processor is only a small percent of the total area, it still consumes 90% of the power. The point is that from a power perspective you are going to have a lot of memory transistors due to power density problems. DRAM has a lower power consumption per $mm^2$ (lower PDA) due to multiple voltages, worse transistors, etc. (Most power consumption is in the sense-amplifiers and not in the bit array). The difference between DRAM/logic power consumption is 10x, even with high use 5x. Should one spread logic around to lower PDA hot spots? Perhaps implement more logic but not use it all the time?

# Author Index